DAYANANDA SAGAR COLLEGE OF ENGINEERING
(An Autonomous Institute affiliated to VTU, Belagavi, Approved by AICTE & ISO 9001:2008 Certified)
Accredited by National Assessment & Accreditation Council (NAAC) with ‘A’ grade, Shavige Malleshwara Hills, Kumaraswamy Layout, Bengaluru-560078.

INTERNSHIP REPORT
On
“Arduino to Micro Python Transcompiler”

Submitted By
Ashutosh Pandey
1DS17CS026
[Eighth Semester B.E (CSE)]
2020-2021

Under the guidance of
Dr Vindhya Malagi
Associate Professor
Dept. of CSE
DSCE, Bangalore

Department of Computer Science and Engineering
Dayananda Sagar College of Engineering
Bangalore-78

VISVESVARAYA TECHNOLOGICAL UNIVERSITY, BELGAUM

DAYANANDA SAGAR COLLEGE OF ENGINEERING

Shavige Malleshwara Hills, Kumaraswamy Layout, Bangalore-560078

Department of Computer Science & Engineering

<u>CERTIFICATE</u>

This is to certify that the seminar work entitled **"Arduino to Micro Python Transcompiler"** is a bonafide work carried out by Ashutosh Pandey [1DS17CS026] in a partial fulfilment for the 8th semester of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the year 2020-21. The internship report has been approved as it satisfies the academic requirements in respect of Seminar Work prescribed for Bachelor of Engineering Degree.

Signature of Examiners with date **Signature of HOD**

1.
2.

Acknowledgement

I started my journey into the field of Compilers and Linux due to the courses I took at college. I would like to thank my Guide, Dr Vindhya M. for her constant support and supervision throughout. I would like to thank our Vice Principal, Dr D R Ramesh Babu, for his constant support for all my extracurricular endeavors. Finally, I would like to thank our Principal, Dr C.P.S Prakash, for providing me with an opportunity to present this work and the great platform that is Dayananda Sagar College of Engineering. I am much indebted to the Department of Computer Science and Engineering, DSCE, for all the support they have given me and all the faith they have in my abilities.

Finally, I would like to always thank my family for their unfailing help.

Ashutosh Pandey
1DS17CS026

Abstract

The Arduino ecosystem has 2700+ libraries written in the 'Arduino dialect' of C++. For the Portenta to gain widespread MicroPython usage it is required that:

- Many these libraries be ported to MicroPython.
- This porting must be efficient to do, to avoid time delays and unnecessary effort.

The project idea aims to make a *source-to-source* compiler (known as a Transpiler) which can:

- 'Translate' a given example program of a library (in Arduino) to MicroPython automatically.
- Highlight any errors and parts of the code that it cannot translate effectively.
- Be easy to maintain as support for more library code is added down the road.

Technical Details

1.1 Rationale behind approach

Writing a compiler is significantly more work than writing a simple script to mechanically convert one language syntax to another. This particular approach was chosen because of the following reasons:

1. **It is far more flexible:** An equivalent Python (or any other language script) would be long, hard to maintain and difficult to modify without affecting the rest of the program. Moreover, a compiler is designed with a seperate front end that is language agnostic, so the AST (Abstract Syntax Tree) generated by the front end can be used to generate code for target languages other than MicroPython (such as JerryScript).
2. **A compiler handles ambiguity better:** C++ can get ambiguous (an input can be interpreted multiple ways) and context sensitive (the meaning of a particular statement depends on the words before it and after it).

For example: The * operator can mean multiplication as in a*b or a pointer in int *a. If we write a C or Python script, we will have to include special cases for all such instances such as *, unary minus etc. Compilers can be designed with *lookahead*, so they can take into account the next few characters of an input and assign meaning accordingly. There are many such methods such as LALR parsing, GLR parsing etc.

3. **MicroPython is constantly evolving:** MicroPython is still growing and adding new features. The latest release was less than 60 days ago. This makes it essential for our project to be amenable to change, which is easier with a Compiler than a handwritten script. To add a new keyword, we just need to make an entry to the symbol table and the compiler will do the rest. Since our implementation language is C, which all Arduino developers are familiar with, this also makes the compiler easier to maintain.
4. **Better target code optimisation:** The equivalent MicroPython code is much more compact than the given Arduino code for most applications. Compilers are better suited to implement optimisation algorithms and processes like peephole optimisation.

1.2 Feasibility & Scope

Given the large scope of the project it becomes necessary to define boundaries of what is doable within the GSoC time period, while keeping in mind to have the project be useful enough to be helpful to the Arduino community long after the GSoC coding period is over.

1. **Working with the Arduino Reference:** Unlike the libraries which have a lot more C++ going on, the examples for each library itself are given in a lot more concise, easy to parse and understand 'Arduino language'. This is not by accident, since the language after all is designed to be easy to understand for beginners. The Arduino Reference defines around 200+ keywords which is a good starting point to tokenize the program input, perform semantic analysis, generate the AST which can then be transformed into the AST of the target language, and then the target language

(MicroPython) itself. The MicroPython website also lists equivalent libraries, so finding equivalent keywords and functions is not difficult.

2. **Building a generic C++ parser is very difficult:** It would ideally be a lot better if we could parse the whole C++ language instead of just Arduino, but there has been research that states that "C++ grammar is ambiguous, context-dependent and potentially requires infinite lookahead to resolve some ambiguities". It makes far more sense to limit ourselves to Arduino.

3. **Successful transpiler projects exist** C++ has strict typing and has a lot more details about the size of memory allocated and resource management. So when we translate C++ code to MicroPython, no assumptions have to made. This is not true when we go from Python --> C++. Moreover, a number of successful transpilers such as Coffeescript, TypeScript, Babel are in widespread use. The Haxe project compiles to multiple target languages. Even a tool to convert C++ code to Python exists (see: pybee/seasnake).

4. **The goal is to translate as many library examples as possible:** Even if 80% or only 60% of a given example program can be translated, it still increases the speed of porting libraries over writing everything by hand.

1.3 Implementation Details

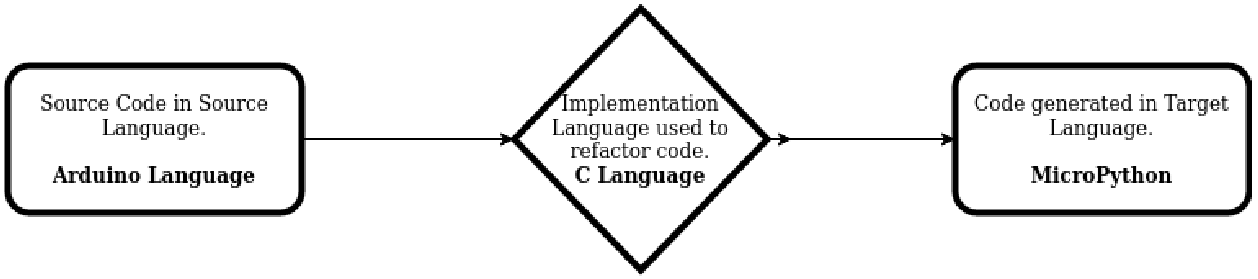
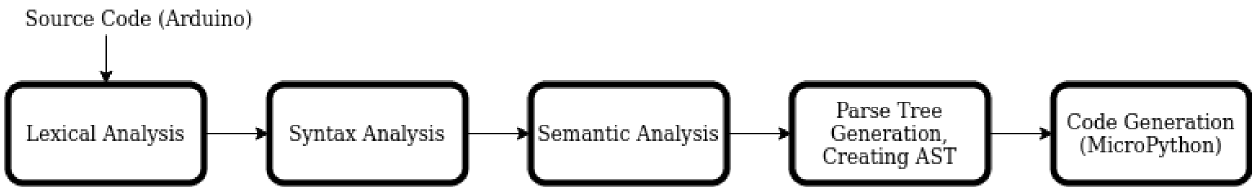

Source Code in Source Language.
Arduino Language
Implementation Language used to refactor code.
C Language
Code generated in Target Language.
MicroPython

Fig 1: Transcompiler Implementation

A Transpiler is basically a compiler in which the source language is converted into equivalent high level (human readable) code in another language instead of generating low level byte code or machine code. Our Transpiler consists of two parts:

- **The Front end:** This stage has to split the input into tokens, analyse the syntax, generate a parse tree and ultimately the Abstract Syntax Tree (AST).
- **The Back end:** To generate the target language (MicroPython).

1.3.1 Overall Compiler Design



I used Practical Approach to Compiler Construction as a reference while outlining the transpiler design.

1. **Source Code:** The Arduino sample code. It may be from inbuilt libraries, external libraries or even self written code.
2. **Lexical Analyser:** The lexical analyser is a program that takes a file (or program) as input and gives the tokens (basic units) of the given code as output. The lexical analyser must be able to identify numbers, identifiers, keywords, special characters etc. Ordinarily comments are ignored, because they are not significant to the program itself. In our program we will choose to keep the multiline comments** /* ....*/** because they usually have important details pertaining to the program (such as the creator's name and program description) while removing the single line **//** comments as they wont make much sense after the program is translated. I will be using **Flex** to create a lexical analyser as it is popular, well documented and open source.
3. **Syntax Analysis:** The syntax analysis phase also groups together the tokens generated by the above lexical analyser according to the syntax rules of the source language (Arduino). The syntax

tree is constructed in this phase. For this phase I use **Bison** for the same reason I used Flex above, in addition to Bison having very good compatibility with Flex.

4. **Semantic Analysis and AST creation:** We need to perform type checking, traversing the tree and adding the right information to the abstract syntax tree is done at this stage.
5. **Code Generation:** We traverse through the AST created in the above steps, ordinarily most compilers create an 'Intermediate Representation' (IR) language at this stage which can be converted to byte code or machine code, but since we don't need machine code, we'll be using MicroPython as our IR language directly and converting the AST to MicroPython statement by statement linearly. Since this compiler produces source code which will be read by a human, we need to worry about readability and 'correctness' as much as the transpiler can manage.

1.3.2 Lexical Analysis

Let us take a snippet from an Arduino program, I chose the 'BarGraph' example from File--> Examples --> Display --> BarGraph in the Arduino IDE which has this typical setup() :

```
void setup() {
  // loop over the pin array and set them all to output:
  for (int thisLed = 0; thisLed < ledCount; thisLed++) {
    pinMode(ledPins[thisLed], OUTPUT);
  }
}
```

ideally we want our lexical analyser (**lexer**) to give the output as something like this:

void (keyword), setup (keyword), for (keyword) , (,) , { , } , 0 (integer constant) , // (comment line) , thisLed (identifier) , = , < , ++ , pinMode (Keyword)

- From the above we can see that the Lexical Analyser should be able to differentiate between variable names that the programmer gives (like **thisLed**) and words that have special meaning to the compiler like **setup() **, for, **pinMode** etc. We do this by listing a set of reserved words in the language specification, telling the transpiler to take special care of them. Luckily, Arduino provides us a good list of the keywords we need to have in mind in their language reference. In our transpiler, we will be keeping the following words as keywords:

	Arduino			
digitalRead()	isAlpha()	Serial (All)	int	# define
digitalWrite()	isALphaNumeric()	Stream (All)	long	# include
pinMode()	isAscii()	Keyboard (All)	short	block comment
analogRead()	isControl()	Mouse (All)	size_t	single line cmt
analogReference()	isDigit()	Floating pt const (All)	string	semicolon
analogWrite()	isGraph()	Integer Const (All)	unsigned char	curly braces
analogReadResolution()	isHexadecimalDigit()	HIGH/LOW	unsigned int	remainder
analogWriteResolution()	isLowerCase()	INPUT/OUTPUT/PULLUP	unsigned long	multiplication
noTone()	isPrintable()	LED_BUILTIN	void	addition
pulseIn()	isPunct()	true	false	word

	Arduino			
pulseInLong()	isSpace()	(unsigned int)	const	division
shiftIn()	isUpperCase()	(unsigned long)	scope	assignment
shiftOut()	isWhitespace()	byte()	static	not equal to
tone()	random()	char()	volatile	less than
delay()	randomSeed()	float()	PROGMEM	less than equal to
delayMicroseconds()	bit()	int()	sizeof()	equal to
micros()	bitClear()	long()	loop()	greater than
millis()	bitRead()	word()	setup()	greater than equal to
abs()	bitSet()	String()	break	logical not
constrain()	bitWrite()	array	continue	logical and
map()	highByte()	bool	do..while	logical or
max()	lowByte()	boolean	else	reference operator
min()	attachInterrupt()	byte	goto	dereference

	Arduino			
pow()	detachInterrupt()	char	if	bitwise and
sq()	interrupts()	double	return	bitwise left
sqrt()	noInterrupts()	float	switch..case	bitwise right

- The above is our keyword table. Please note that words with **(All)** suffixed have multiple functions inside them, like the Keyboard and Mouse. All of the functions in these will be implemented.
- Since there are tokens such ++ (increment) and >= (greater than or equal to) we have to take special care to not interpret these as + + or as < followed by = respectively. For this we will have to implement *lookahead *in our lexer.
- We need to implement the lexer to identify the following tokens: Identifiers, character constants, floating point nubers, operators, strings, keywords, even comments have to be parsed so that they can be ignored (In our case, mainly the single line comments).
- We need to keep track of the indentation in the code to get a properly indented output, to do this we must note the beginning and end of the indented block and generate "start block" and "end block" tokens.
- Finally, when the flex scanner returns a stream of tokens, there are two part in the output: the token itself, and a corresponding value. eg: we can set *add *to mean 258, *multiply* to mean 259 etc. This will help us keep track of the tokens later on.

1.3.3 Using Flex lexical analyser

Flex is used for tokenizing especially because it is much more powerful than writing an implementation C or any other language. For instance, to write a basic lexical analyser in C (which can identify identifiers, operators and about 10 keywords) can be found on this site . This tokenizer is over 160 lines of code, is inflexible and has no lookahead (unless we manually implement it through loops).

In contrast, here is a tokenizer that does the same thing:

```
letter      [a-zA-Z]

digit       [0-9]

letter_or_digit [a-zA-Z0-9]

white_space       [ \t\n]

other                   .

%%

{white_space}+              ;

{letter}{letter_or_digit}*          return 1;

{digit}+                                     return 2;

{other}                                      return 3;



%%

int main() {

 int lextoken;

 while (lextoken = yylex())

        printf("%d - %s\n",lextoken, yytext);

}

int yywrap()

{

return 1;
```

}

- Flex is much more compact because it uses 'regular expressions' (popularly shortened to regex). Using the combination of rules given to generate regex in flex, we can define even complex tokens such as floating point numbers in the IEEE standard, which can be represented in the following way:

```
/* float exponent */ EXP ([Ee][-+]?[0-9]+)
//and then using the definition of EXP
/* decimal float */ ([0-9]*.[0-9]+|[0-9]+.){EXP}?[flFL]? [0-9]+{EXP}[flFL]?
```

- The above expression can handle all kinds of float input, like 12 (no digits after decimal), 5.1 , 6.23142 etc.
- As we mentioned above, we have to parse comments so that we can choose to keep them or ignore them. Even this is possible to do in one line:

```
/\ *([^**]|*+[^/ *])*\ *+/
```

- Fortunately for us our source language is mostly C/C++ code so we do not need to worry about such special inputs, they have multiple implementations and we need to just re-implement the ones we need.

Moreover, Flex allows us to use our own custom C code inside the program, so it offers all the customisability of C without any downsides. The generated executable C program is stored in lex.yy.c .

The program can be executed with instructions:

```
 lex filename.l
 gcc lex.yy.c
 ./a.out
```

- As mentioned before, we cannot just take the tokens and store them, there is information associated with tokens such as:

    - Data types and names, for variables.
    - Declaration procedures.
    - Details about parameters (call by value and call by reference).
    - Number and type of arguments passed to functions.
    - All the other keywords that we mentioned in the table above.

- To store all this information, we need to have a **symbol table**. A symbol table can be implemented using list, linked list, binary search tree or hash table. For its speed and effectiveness, and because we will have a rather large number of entries in the table, we use the** hash table**. The symbol table is called using the symbol table routine in the code section of the above lex program.

We can use any Hash function with the Symbol table, usually the source symbol is itself used as a key for the hash function.

Any hash function can be used, so we use the one from Flex and Bison O'Reilly book:

Quoting the author:

The hash function is also quite simple: For each character, multiply the previous hash by 9 and then xor the character, doing all the arithmetic as unsigned, which ignores overflows. The lookup routine computes the symbol table entry index as the hash value modulo the size of the symbol table, which was chosen as a number with no even factors, again to mix the hash bits up.

Here is the Hash function:

```
/* hash a symbol */
static unsigned
symhash(char *sym)
{
unsigned int hash = 0;
unsigned c;
while(c = *sym++) hash = hash*9 ^ c;
}
return hash;
struct symbol *
lookup(char* sym)
{
struct symbol *sp = &symtab[symhash(sym)%NHASH];
int scount = NHASH;
/* how many have we looked at */
while(--scount >= 0) {
if(sp->name && !strcmp(sp->name, sym)) return sp;
if(!sp->name) {
sp->name = strdup(sym);
sp->reflist = 0;
return sp;
}
/* new entry */
if(++sp >= symtab+NHASH) sp = symtab; /* try the next entry */
}
fputs("symbol table overflow\n", stderr);
abort(); /* tried them all, table is full */
}
```

The** lookup** routine takes a string and returns the address of the table entry corresponding to that name, and it creates one if it doesn't exist already. Since the hashing function is being applied on the symbol itself, **strdup** is called to save a copy of the input string (symbol) itself so that it can be entered in the right place in the symbol table entry.

Once we have defined the expressions in flex to handle all possible input and populated the symbol table with all the keywords and tokens, and verified that the Lexer is giving the right output, the Lexical analysis part is done.

We can and should point out some errors at this stage itself. flex will throw an error message along with the line number if the input does not make sense (such as entering a number like 12.3.1) . However we are limited to only reporting basic grammatical error at this stage because the lexical analyser can only handle 1 token at a time. So if we have an input like:

```
for if 1
```

or

```
if while int
```

will still evaluate to true because we are still not capable of capturing context or the true 'meaning' of these tokens in a statement.

Moreover something like:

```
while (digitalRead(buttonPin) == HIGH) {
   calibrate();
  }
  // signal the end of the calibration period
  digitalWrite(indicatorLedPin, LOW);
```

Cannot be evaluated properly because regular expressions cannot handle balancing tokens like parenthesis properly. For that, we will have to analyse the syntax, so we move on to the **syntax analysis** phase.

1.3.4 Syntax Analysis

The basic purpose of parsing is to group the tokens we generated in the lexical analysis phase and then checking them against the rules of the language. things like defining the scope, typing and the way the tokens are put together are done here. Ultimately, the goal of the syntax analysis phase is to generate a **parse tree**.

To illustrate this point, we take a snippet from ForLoopIteration example. It can be found in File-->Examples-->Control-->ForLoopIteration

```
for (int thisPin = 2; thisPin < 8; thisPin++) {
    // turn the pin on:
    digitalWrite(thisPin, HIGH);
    delay(timer);
    // turn the pin off:
    digitalWrite(thisPin, LOW);
  }
```

A psuedocode representation of the output of parsing the** for loop** should look something like this: STATEMENTLIST - Statement list, this statement: | KEYWORD - Assign to word, keyword is: | | ASSIGN - Assign to variable at offset 2, expression is: | | | LESSTHAN - lesser than, lhs is: thisPin | | INCREMENT - Increment operator, lhs is: thisPin STATEMENTLIST - Statement list, continued. . .

The **graphical representation** of the syntax tree generated by the above code block would be:

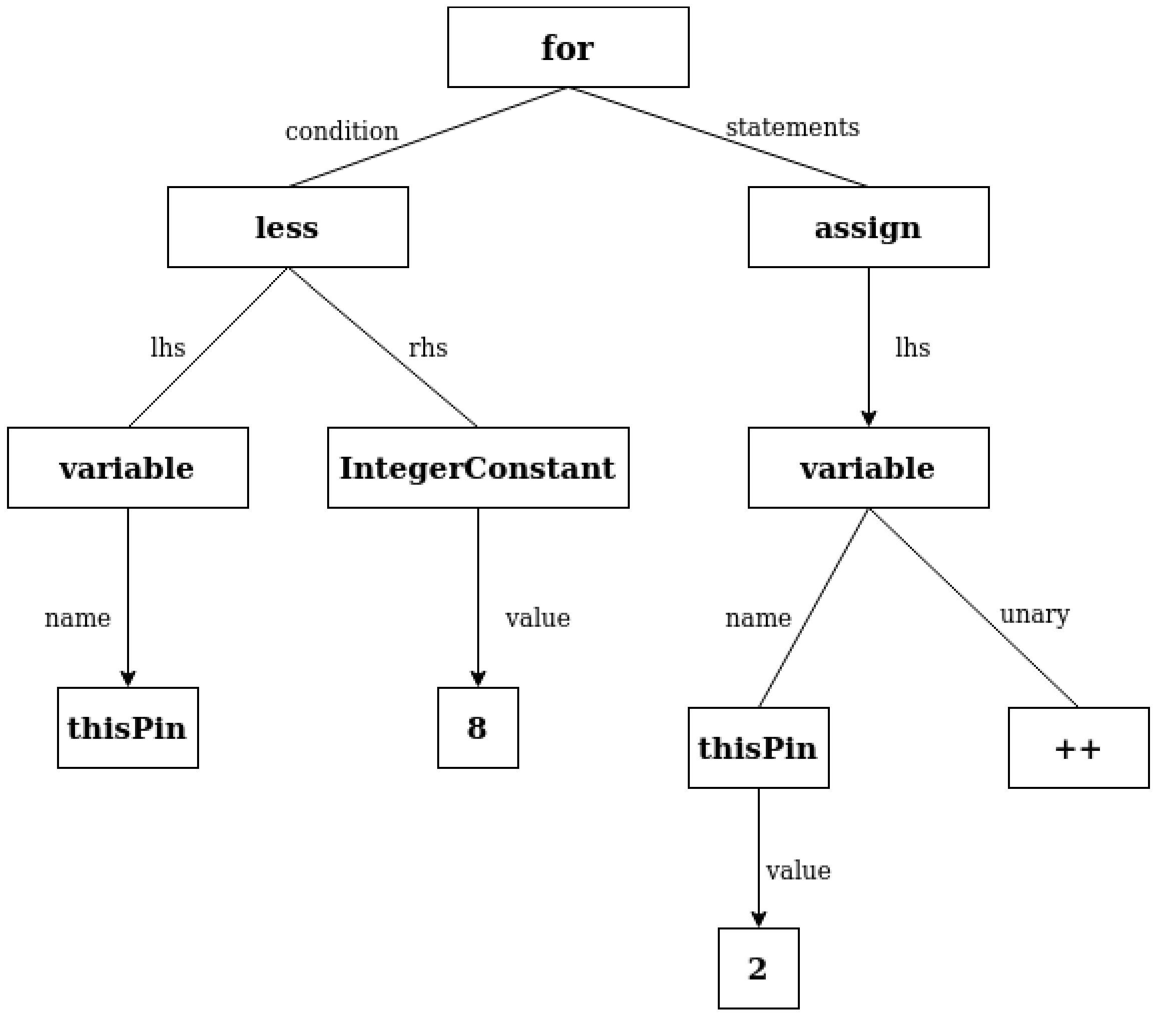


Fig 3: Syntax Tree representation

A few points have to be kept in mind while designing our syntax analyser to generate a parse tree.

- The rules of the grammar must be such that only one such parse tree is possible for any given input. If the maker of the original Arduino code does not take care while writing the code, the translation may result in unpredictable behaviour. When this happens we say the grammar is ambiguous. As mentioned before, some of the ambiguity can be removed by adding one character lookahead so that we don't get errors. Without lookahead it is possible that:

1234+ 12 may result in an error because integer tokens should have no signs like +, - after the token, but if our parser can 'look ahead' by 2 tokens, it will realise that the term is just a part of the larger expression.

- It is possible to parse both from the right and left. Since Left to Right parsers are more common, and when combined with Lookahead are sufficiently powerful for our purposes, we **will use a LALR parser**. Generalised Look Ahead (GLR) parsers are even more powerful, but they increase code complexity by quite a bit and LALR will suffice for our needs.
- The parsing can be done from the root node onwards, considering the alternatives until it reaches the child nodes, or we could start 'bottom up'. Although top-down parsers have their own advantages when we right a parser by hand, we are using **Bison**, which has support for bottom - up parsing. Bottom Up parsing also has the advantage of being more powerful than top down parsers. So we use a **bottom up** approach.

Generating good error messages

Atleast in the initial stages of transpiler development, the support for various library functions will not be complete. Even after all of the core Arduino keywords are integrated, all of the sensors have their own libraries and each of those libraries have their own functions.

Here is a sample code for DHT11 sensor, which is a common temperature and humidity sensor used with the Arduino:

```
#include **<dht.h>


dht DHT;


#define DHT11_PIN 7


void setup(){
  Serial.begin(9600);
}
```

```
void loop()

{

  int chk = DHT.read11(DHT11_PIN);

  Serial.print("Temperature = ");

  Serial.println(DHT.temperature);

  Serial.print("Humidity = ");

  Serial.println(DHT.humidity);

  delay(1000);

}
```

- In the given example, dht.h , DHT.temperature, DHT.humidity etc will not be recognised properly by the transpiler because these are not a part of the core language reference. We need to parse through these examples and point out all the errors at once, so that the programmer can edit those parts with the corresponding MicroPython library and generate a working program.
- `  for (int thisPin = 2; thisPin < 8; thisPin++)`
- `              // turn the pin on:`
- `              digitalWrite(thisPin, HIGH);`
- `              delay(timer);`
- `              // turn the pin off:`
- `              digitalWrite(thisPin, LOW);`
- `            }`
- In this code snippet given above, the ** { ** is absent on the first line. But the error may not be reported until the last line. It is tempting to try and code a fix for this into the transpiler, but we should avoid designing a compiler to fix such bugs as it may introduce new problems into the

code. Our best bet is to report the line number accurately and hope that the programmer can fix it themselves.

- void yyerror(char *s)
- {
- printf("%s on line %d\n",s,yylineno);
- }
- Fortunately, Bison comes inbuilt with a **yyerror** function and **yylineno** to handle errors and to note the line number. Otherwise we would have to include code to count line numbers and report errors ourselves. However we still have the option of generating custom error messages for particular kinds of errors.

1.3.5 Using Bison for Parsing

Just like with Flex, it is possible to write the whole parser in C or any other programming language. But writing Bottom up LALR(1) parser is far from trivial. Given that we are trying to parse a big chunk of C/C++, it makes sense to use Bison. Handwritten parsers may give ambiguous output if not designed properly, this is not a problem with Bison since it *simply will not parse an ambiguous grammar*. This saves us a lot of pain while debugging our transpiler later.

A Bison program usually has 3 parts, just like a lex program.

%{ C Declarations %} Bison Declarations %% Grammar Rules %% Additional C Code

In the C declarations part, we can call the yylex() function that we generated earlier. we can call yyerror() also. So Bison part of the transpiler will get tokenized input, this approach makes our transpiler modular. It is also possible to write the yylex() function inside the bison program itself, but this will make our code harder to debug. So we call yylex() from the Bison program.

The GNU Bison manual is a great resource and I will be making use of it while writing the parser.

- We will name terminal symbols with C like Identifiers to make them easier to remember. symbols like + - * / ; , etc. The Syntax for this is:
- stmt: RETURN expr ';' ;
- //for the return character to punctuate rules
- statements like x+1, x+4211 etc are equivalent, but we need to take care that constants are recognised as such, so tokens have two components:
    - Type: INTEGER, IDENTIFIER etc.
    - Semantic value: the value, name etc.
- Finally, we rules. Just parsing the input is not enough. It has to produce output based on input if we have defined rules for that particular input. If we allow two floating point numbers to be added in Arduino for example, we can use:

expr: expr '+" expr ($$ = $1 +$3;} ;

The Appel Modern Compiler in C book, popularly known as the tiger book has a great set of examples on Flex and Bison based calculators.

- The error recovery problem that we mentioned earlier is easy to resolve. The Bison language reserves the word error. yyerror() can be used to make the program wait, but to make sure that it continues parsing (so that we get a list of all potential errors at once like normal compilers).
- line:
- '\n'
- | exp '\n'   { printf ("\t%.10g\n", $1); }
- | error '\n' { yyerrok;   }
- ;

The code snippet above does this. yyerror() is still called eventually, so that we can display our relevant error messages;

- Finally when we have defined all the rules for our language and the parser is built, we can turn our attention to the output of the program. Just like how the lexer gave us the yylex.c file as an output, Bison gives a parser implementation file that can be used to implement the output. This file is called yyparse().

Generation of a Parse tree

After the language has been parsed, we need to construct a parse tree like the one we saw in figure 3. This is not very difficult to do. We need to judge the number of unique 'nodes' in our expression and just modify the code to incorporate the tree datastructure by defining the nodes. Then the expressions, instead of being evaluated will generate a tree.

- To generate the tree, first we have to design it We need to specify all the different node types and what each node should contain. The node for int will have size, type, name while a token like semicolon (;) does not have that many details.
- It might be possible to perform some simplification here. For instance the **MicroPython code** for BMP180 (Temperature and Barometric pressure sensor) on an ESP32 is:
- from bmp180 import BMP180
- from machine import I2C, Pin   \# create an I2C bus object accordingly to the port you are using
- \#bus = I2C(1, baudrate=100000)   \# on pyboard
- bus = I2C(scl=Pin(22), sda=Pin(21), freq=100000)   # on esp8266
- bmp180 = BMP180(bus)
- bmp180.oversample_sett = 2
- bmp180.baseline = 101325
-
- temp = bmp180.temperature

- p = bmp180.pressure
- altitude = bmp180.altitude
- print("Temperature: ",temp)
- print("Pressure: ",p)
- print("Altitude: ",altitude)

The corresponding **Arduino code** for BMP180 to measure temperature and pressure is:

```
\#include <Wire.h> //Including wire library


\#include <SFE_BMP180.h> //Including BMP180 library


\#define ALTITUDE 35.6 //Altitude where I live (change this to your altitude)


SFE_BMP180 pressure; //Creating an object


void setup() {
 Serial.begin(9600); //Starting serial communication


 Serial.println("Program started");


 if (pressure.begin()) //If initialization was successful, continue
  Serial.println("BMP180 init success");
 else //Else, stop code forever
 {
  Serial.println("BMP180 init fail");
```

```
    while (1);
  }
}

void loop() {
  char status;
  double T, P, p0; //Creating variables for temp, pressure and relative pressure

  Serial.print("You provided altitude: ");
  Serial.print(ALTITUDE, 0);
  Serial.println(" meters");

  status = pressure.startTemperature();
  if (status != 0) {
    delay(status);

    status = pressure.getTemperature(T);
    if (status != 0) {
      Serial.print("Temp: ");
      Serial.print(T, 1);
      Serial.println(" deg C");

      status = pressure.startPressure(3);
      if (status != 0) {
        delay(status);

        status = pressure.getPressure(P, T);
        if (status != 0) {
          Serial.print("Pressure measurement: ");
          Serial.print(P);
          Serial.println(" hPa (Pressure measured using temperature)");

          p0 = pressure.sealevel(P, ALTITUDE);
          Serial.print("Relative (sea-level) pressure: ");
          Serial.print(p0);
          Serial.println("hPa");
        }
      }
    }
  }
  delay(1000);
}
```

- The first thing we can see is even after removing a lot of the comments and syntactic sugar is that the MicroPython code is much shorter. Infact it does not even have a setup() or loop() method unlike all Arduino programs where it is mandatory. As we work with the Portenta board running MicroPython, we can conclude what does not need to be translated and just ignore those parts of the input file. This will simplify the abstract syntax tree and make code generation easier.

- After we produce the abstract syntax tree, we are near the end. The AST contains only the important parts of the source language that we need to translate into the target language. Unlike parse trees of which there can be only one for any given grammar, we can have different types of ASTs for the same grammar depending upon the level of optimisation we are doing to the source code. A sample AST for the Syntax tree we generated in section 1.3.5 is given below:

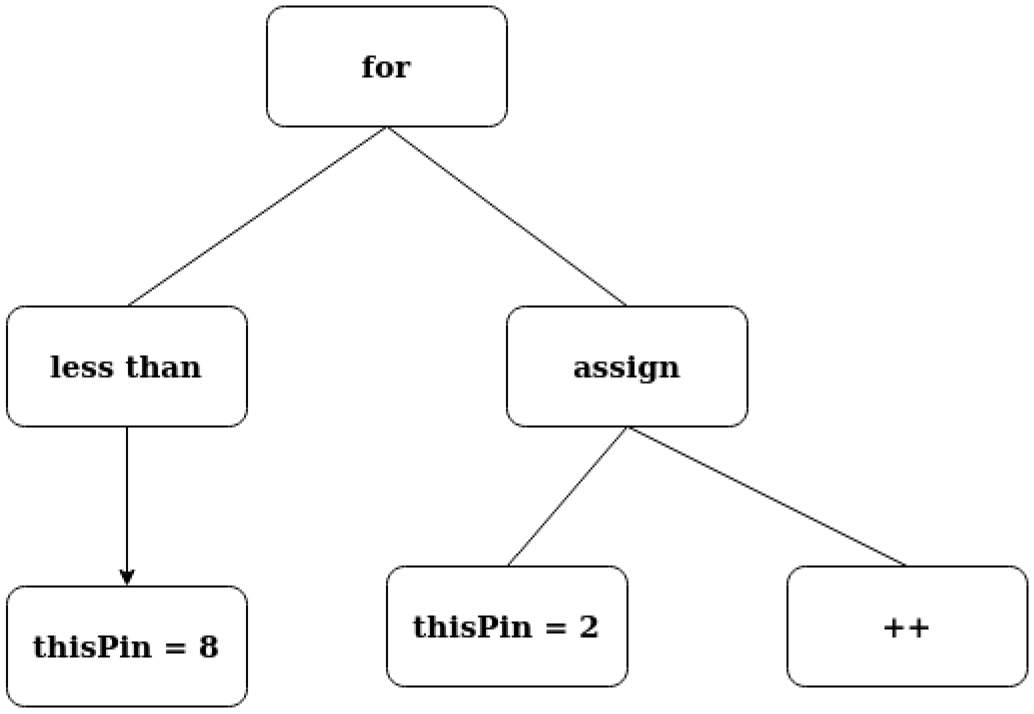


Fig 4: Sample AST

1.3.6 Semantic Analysis and Target Code

The semantic analysis phase is the last phase that is concerned with the source language (Arduino). We now have an Abstract Syntax tree produced by the Syntax analyser, converting the AST into the source language is a simple matter of 'flattening' the tree and translating the Arduino code into MicroPython statement by statement. Since we are compiling from source to source, our back end is rather simple compared to LLVM which has millions of lines of code, we don't really have to worry about machine specific details either, because we are not generating machine (or in Arduino's case, Intel Hex format code).

In the semantic analysis phase we have to take care of context sensitive analysis. Modern high level languages (and Arduino is no exception) have statements that are ambiguous on their own, but are perfectly clear once the program is read as a whole. I gave an example of this in 1.1 with C++ having potentially ambiguous syntax. There is even a Wikipedia article about the Most Vexing Parse in C++ .

The Abstract Syntax Tree generated in the previous phase is missing some key details that are necessary for it to translate well to MicroPython. Most of these details concern the management of names and types. A lot of the details are stored in the symbol table we created earlier and we can use that symbol table here to annotate the AST. We will consider them one by one. After all these steps are performed, we have an annotated syntax tree which can then be 'walked' to get the language.

- **Type and Type Checking:** C++ (Arduino) is a strongly typed language which means it places strict limits upon how much memory and resources are allocated at compile time itself. MicroPython is also built for microcontrollers, and has strong typing itself. Arduino also allows safe type conversions, as does MicroPython.

*** Type Information:** typedef and structures allow custom datatypes and it can get potentially troublesome to store data about them and their handling, fortunately, they are not used too often in microcontrollers with limited memory and processing power. Same goes for multidimensional arrays.

*** Type Checking:** Python allows for dynamic typing, allowing variables to be declared and type checking happens during runtime. This can be troublesome. However we can get around this problem since we are translating existing Arduino code to MicroPython. Since the sensors and data coming from them is unchanged, it is reasonable to assume the data types used in the Arduino code can be used in the resulting Python code too.

To preserve the type information, we label the nodes of the AST with type information.

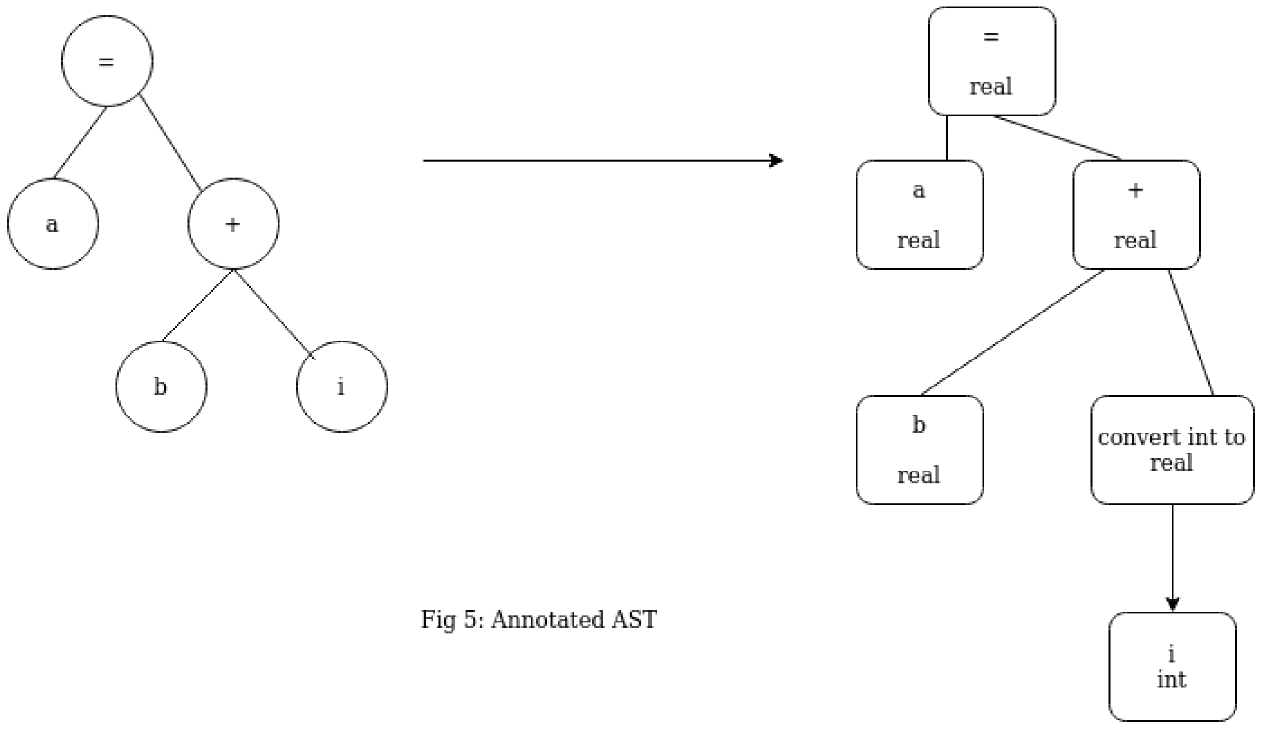


Fig 5: Annotated AST

In the above example variable **i** is promoted to real because **b** and **a** are real.

- **Type Rules:** For microcontroller applications where date and time are involved (like the DS1307 RTC module which keeps track of time) and we need to measure something every x minutes, hours or days, we need to determine the rules for adding and subtracting units of time.
- **Scope information:** In Arduino {} are used to denote scope of a variable, function or loop. In MicroPython, we use indentation instead of braces. The compiler has to store tokens corresponding to the beginning and end of blocks and then generate appropriate tab spacing to solve this issue.

Target Code generation

Most compilers at the end of the semantic analysis phase convert the annotated AST to a IR (Intermediate representation) language. The IR is then converted to machine code. We use linear substitution to generate MicroPython code from the annotated AST. Our annotated AST contains tokens, their details, type, function etc. There are not many resources on transpiler design, but most sources suggest that a 1:1 substitution from source language elements to target language elements should be enough.

1.3.7 Testing and creating examples for Portenta

After the transpiler design is completed, the next step is to 'debug' the generated code using the Portenta board. The end goal is to produce code that can compile with as few modifications as possible. I will test out the Portenta with the following sensors that I have on hand:

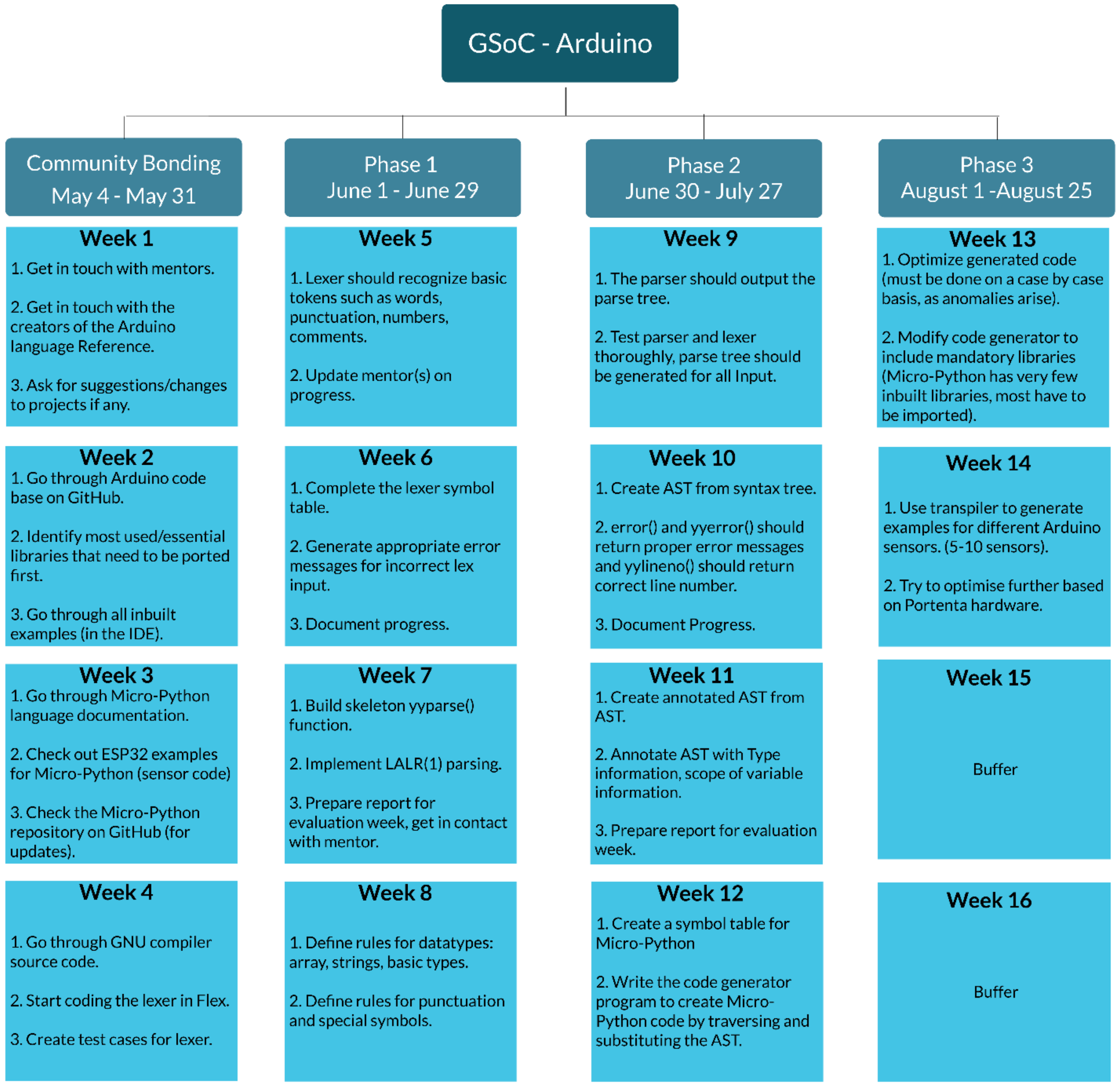
Sensor	**Usage**
BMP280	To measure temperature and pressure
DHT11	To measure relative humidity and temperature
RTC	To measure time
HCSR04	Ultrasonic Sensor
IR Sensor	To detect objects using IR LED and Photoresistor
PIR Sensor	Passive IR, to detect objects
HC05	Bluetooth
ESP01	Wifi
HC12	Long Range Radio Module
433 Mhz Transciever	Short Range Radio module
MPU6050	Accelerometer and Gyroscope IMU
MPU9250	Accelerometer, Gyroscope,Magnetometer IMU
MQ5 and MQ3	Alcohol, Butane, Smoke etc

Sensor	**Usage**
Rainfall Sensor	To detect water droplets
Soil Moisture Sensor	Soil Humidity measurement
Flow Rate sensor To measure flow rate of water	
KY039	To measure Pulse rate
AD8232	To measure ECG of the heart
Myoware Sensor	To measure contraction of muscles
Neo 6M	GPS Module
2.8 inch TFT LCD	Screen
16x2 LCD	LCD Display module
0.96 inch OLED	Small OLED Screen

Note: Even ESP libraries do not exist for some of these sensors, and the library code may require a lot of rewriting, I will attempt to make as many sensors work with the portenta as possible.

II. Schedule of Deliverables

Work Breakdown Structure



GSoC - Arduino

Community Bonding
May 4 - May 31

Week 1
1. Get in touch with mentors.
2. Get in touch with the creators of the Arduino language Reference.
3. Ask for suggestions/changes to projects if any.

Week 2
1. Go through Arduino code base on GitHub.
2. Identify most used/essential libraries that need to be ported first.
3. Go through all inbuilt examples (in the IDE).

Week 3
1. Go through Micro-Python language documentation.
2. Check out ESP32 examples for Micro-Python (sensor code)
3. Check the Micro-Python repository on GitHub (for updates).

Week 4
1. Go through GNU compiler source code.
2. Start coding the lexer in Flex.
3. Create test cases for lexer.

Phase 1
June 1 - June 29

Week 5
1. Lexer should recognize basic tokens such as words, punctuation, numbers, comments.
2. Update mentor(s) on progress.

Week 6
1. Complete the lexer symbol table.
2. Generate appropriate error messages for incorrect lex input.
3. Document progress.

Week 7
1. Build skeleton yyparse() function.
2. Implement LALR(1) parsing.
3. Prepare report for evaluation week, get in contact with mentor.

Week 8
1. Define rules for datatypes: array, strings, basic types.
2. Define rules for punctuation and special symbols.

Phase 2
June 30 - July 27

Week 9
1. The parser should output the parse tree.
2. Test parser and lexer thoroughly, parse tree should be generated for all Input.

Week 10
1. Create AST from syntax tree.
2. error() and yyerror() should return proper error messages and yylineno() should return correct line number.
3. Document Progress.

Week 11
1. Create annotated AST from AST.
2. Annotate AST with Type information, scope of variable information.
3. Prepare report for evaluation week.

Week 12
1. Create a symbol table for Micro-Python
2. Write the code generator program to create Micro-Python code by traversing and substituting the AST.

Phase 3
August 1 -August 25

Week 13
1. Optimize generated code (must be done on a case by case basis, as anomalies arise).
2. Modify code generator to include mandatory libraries (Micro-Python has very few inbuilt libraries, most have to be imported).

Week 14
1. Use transpiler to generate examples for different Arduino sensors. (5-10 sensors).
2. Try to optimise further based on Portenta hardware.

Week 15
Buffer

Week 16
Buffer

0. Before Community Bonding

- The time period from April 1- May 4 is the pre - community bonding phase.
- Objective is to push a good number of Pull Requests to the Arduino GitHub repository.
- Getting familiar with Arduino's processes and codebase.

- Take a course on Compiler design. I went through 4 textbooks (Des Watson, O'Reilly, the 'Dragon Book' and the 'Tiger Book' to write this proposal. I have taken compiler design as a course this semester at my college, yet to be safe it would be good to take one of the MOOCs from Coursera or EdX, and go through MIT OCW notes.

1. Community Bonding Period

1.1 Week 1:

- Contact the Arduino developers, and try to get in touch with the developers who wrote the Arduino Language reference.
- Get in touch with my project mentor, introduce myself, workout a good time to communicate (taking into account timezone differences and their schedule).
- Ask the mentor for any suggestions/changes to the project. The project proposal reflects my knowledge as of writing, there may be essential amendments neccessary and it would be better to make them early.

1.2 Week 2:

- Go through the Arduino code base again. Make notes of which tokens: - datatypes, identifiers, punctuation marks, keywords etc are used most often. Some of them may be functions defined only inside certain libraries, pay special attention to those.
- Go through all the example programs given in the IDE, the inbuilt examples are what we'll be using as a template for a basic Arduino program, so it is essential to get familiar with them.
- Get in touch with other GSoCer's , both within the Arduino community and outside.

1.3 Week 3:

- MicroPython is the target language for compilation and to implement the compiler properly, I have to know the source language and target language equally well. So going through the MicroPython documentation is essential.

- At present the ESP32 is among the only popular boards to have widespread Micropython usage. It also runs Arduino style code from the Arduino IDE, so it makes for good comparison between MicroPython code and C/C++ style code. I will go through the code for various sensors (the ones mentioned above in this proposal, since I have them at my disposal to test).
- Go through the MicroPython code base on GitHub.

1.4 Week 4:

- GNU/GCC is among the most popular compilers for C++. Although the code base has millions of lines and it is impossible to comprehend all of it, I will take a look at it to understand the idiosycracies of compiler design.
- Start coding the lexer. Even though this is before the official coding period starts, I'd like to start working on the lexical analyser a bit in advance so that I can iron out any problems in it. The success of the Bison Parser is very much dependent on the lexer working correctly, otherwise it will become extremely difficult to debug in the later stages.
- Work on creating a comprehensive suite of test inputs for the lexer, to check all corner cases and verify that the lexer is indeed working correctly.

2. Phase 1

2.1 Week 5:

*** Deliverable:** The lexer by the end of week 5 (first week of the coding period) , should be able to recognise basic tokens such as words, punctuation marks, numbers, comments etc.

- This is not very hard to do and since lexical analysers for C code have been implemented before, they can be used as a reference.
- Give the mentors an update on my progress so far and take their feedback.

2.2 Week 6:

- **Deliverable: **Symbol table.

- **Deliverable:** Generating appropriate error messages for incorrect lex input.
- The lexical analyser has a symbol table that stores the keywords and the meaning of different tokens, this symbol table will be used later by the parser and during the creation of the annotated abstract syntax tree. The flex lexical analyser has to report incorrect lexemes.

2.3 Week 7:

- **Deliverable:** Working basic parser in Bison.
- The parser we are implementing is a LALR(1) parser. This feature will be implemented during this time period.
- This is also the week before the phase 1 evaluations, so getting in contact with the mentors to discuss any possible improvements before the week 1 evaluation proceeds.

2.4 Week 8:

- **Deliverable:** Parser rules for datatypes: array, strings, basic types should be defined.
- **Deliverable:** Defining rules for punctuation and special symbols like escape, semicolon, equal to, plus, minus, multiply, divide etc.

3. Phase 2

3.1 Week 9:

- **Deliverable:** Syntax Tree.
- The rest of the week is to test the lexer and parser thoroughly before moving on to the AST and code generation phase.

3.2 Week 10:

- **Deliverable:** Create the Abstract Syntax tree from the syntax tree, by stripping out unneccessary tokens and nodes.
- **Deliverable:** The error() and yyerror() functions should return proper error messages.

3.3 Week 11:

- **Deliverable:** Create annotated AST by using the symbol table we built earlier, the annotated AST should have details about Type information, scope of the variable etc.

3.4 Week 12:

- **Deliverable:** Create the symbol table/list corresponding commands of MicroPython with respect to Arduino. The AST will be traversed to generate the target language.
- **Deliverable:** The basic code generator function must be ready.

4. Phase 3

4.1 Week 13:

- **Deliverable:** Optimisation of the generated code (removing uneccessary code, anomalies).
- Since MicroPython seems to include very few features by default, it may be 'mandatory' to import some libraries (much like Wire.h is for many Arduino programs).

4.2 Week 14:

- **Deliverable:** Library example code for 5-10 sensors.
- We will use the transpiler we built to create some code for libraries, based on Portenta Hardware and the output, further optimisation will almost certainly be needed.

4.3 Week 15:

- Buffer week for unexpected delays.

4.4 Week 16:

- Buffer week for unexpected delays, final submissions for GSoC 2020.

Arduino to MicroPython Transcompiler

Supported conversions (As of 31/08/2020)

I/O: Digital, Analog and Advanced:

- digitalRead(inPin) to Pin.value(inPin)
- digitalWrite(pin, value) to Pin.value(pin, value)

- pinMode(pin, INPUT) to pin.Mode(p(number).IN)
- pinMode(pin, OUTPUT) to pin.Mode(p(number).OUT)
- pinMode(pin, PULLUP) to pin.Mode(p(number).PULL_UP)
- analogRead(analogpin) to ADC.read_u16(analogpin)
- analogWrite(pin, value) to machine.PWM(pin, value)
- pulseIn(pin, value) to machine.time_pulse_us(pin, value)

Note: some of these statements also insert a comment asking the user to import machine module.

Time:

- delay(ms) to utime.sleep_ms(ms)
- delayMicroseconds(us) to utime.sleep_us(us)
- millis() to utime.ticks_ms()
- micros() to utime.ticks_us()

Math:

- pow(base, exponent) to math.pow(base, exponent)
- sqrt(x) to math.sqrt(x)
- cos(rad) to math.cos(rad)
- sin(rad) to math.sin(rad)
- tan(rad) to math.tan(rad)
- Macro PI = 3.14159... to math.pi
- Macro EULER = 2.7... to math.e

Characters:

- isAlpha(thisChar) to ure.match('[A-Za-z]', String)
- isAlphaNumeric(thisChar) to ure.match('[A-Za-z0-9]', String)
- isAscii(thisChar) to ure.match('\w\W', String)

- isDigit(thisChar) to ure.match('\d', String)
- isLowerCase(thisChar) to ure.match('[a-z]', String)
- isPunct(thisChar) to ure.match('\W', String)
- isSpace(thisChar) to ure.match('\f\n\r\t\v\s', String)
- isUpperCase(thisChar) to ure.match('[A-Z]', String)
- isWhitespace(thisChar) to ure.match('\s\t', String)

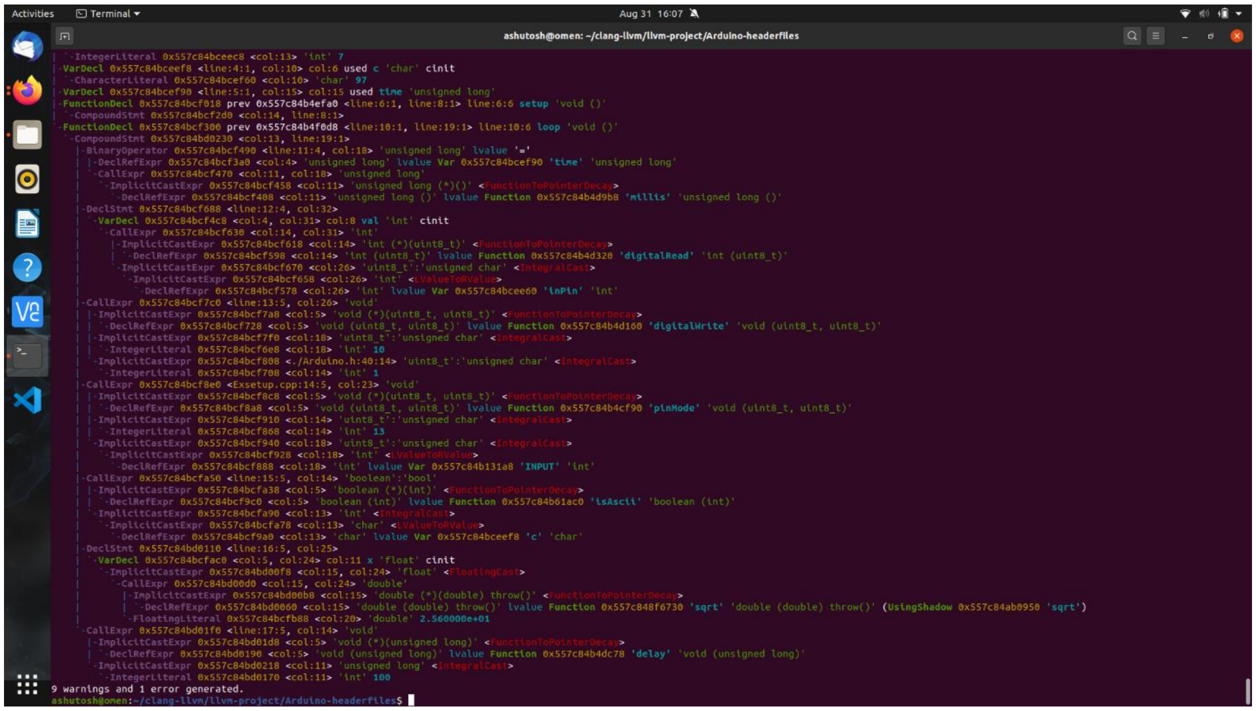
Note: Wherever these expressions occur in the sketch, the tool gives a warning to import ure

Sketch:

- void loop() to While True:
- void setup() is deleted, MicroPython sketches don't have void setup()
- {} are commented out with a '#' since Python does not use curly braces for scope. They are not deleted however, since a user converting a badly indented sketch might lose scope information. So we comment out the {} for every expression where it is a compound statement.
- If() and for() are currently detected by the tool, the if and else part are expanded with comments. More features relating to this will be released in upcoming versions.

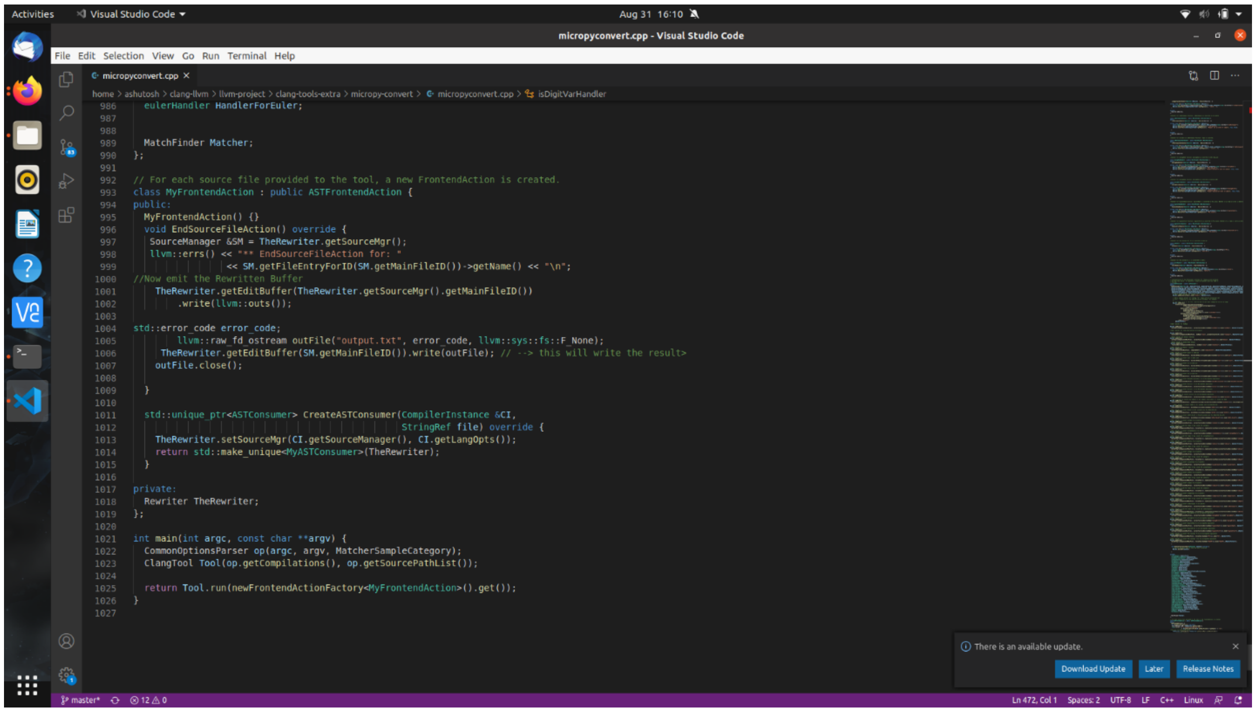
Development Process:

We use Clang to dump the AST to the terminal. Initially, there are a lot of unresolved expressions as Arduino keywords cannot be parsed by Clang. So we include all the Arduino core files in one folder, along with all the relevant AVR-libC headers that Arduino requires. This is because we want the AST, and not the hex dump. After editing the headers to resolve dependency issues and some definitions (such as renaming macros to const variables so that they can be seen by the compiler), we get an AST as shown below:



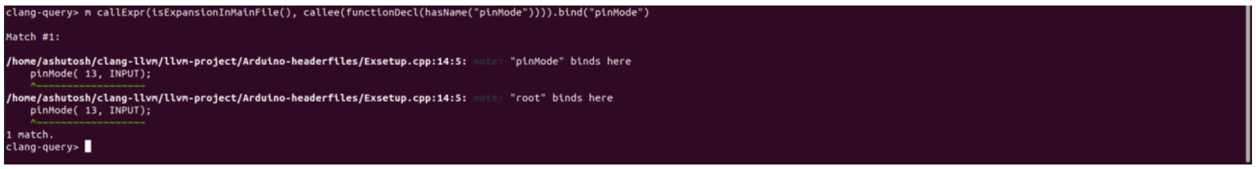
```
| `-IntegerLiteral 0x557c84bceec8 <col:13> 'int' 7
|-VarDecl 0x557c84bceef8 <line:4:1, col:10> col:6 used c 'char' cinit
| `-CharacterLiteral 0x557c84bcef60 <col:10> 'char' 97
|-VarDecl 0x557c84bcef90 <line:5:1, col:15> col:15 used time 'unsigned long'
|-FunctionDecl 0x557c84bcf018 prev 0x557c84b4efa0 <line:6:1, line:8:1> line:6:6 setup 'void ()'
| `-CompoundStmt 0x557c84bcf2d0 <col:14, line:8:1>
`-FunctionDecl 0x557c84bcf300 prev 0x557c84b4f0d8 <line:10:1, line:19:1> line:10:6 loop 'void ()'
  `-CompoundStmt 0x557c84bd0230 <col:13, line:19:1>
    |-BinaryOperator 0x557c84bcf490 <line:11:4, col:18> 'unsigned long' lvalue '='
    | |-DeclRefExpr 0x557c84bcf3a0 <col:4> 'unsigned long' lvalue Var 0x557c84bcef90 'time' 'unsigned long'
    | `-CallExpr 0x557c84bcf470 <col:11, col:18> 'unsigned long'
    |   `-ImplicitCastExpr 0x557c84bcf458 <col:11> 'unsigned long (*)()' <FunctionToPointerDecay>
    |     `-DeclRefExpr 0x557c84bcf408 <col:11> 'unsigned long ()' lvalue Function 0x557c84b4d9b8 'millis' 'unsigned long ()'
    |-DeclStmt 0x557c84bcf688 <line:12:4, col:32>
    | `-VarDecl 0x557c84bcf4c8 <col:4, col:31> col:8 val 'int' cinit
    |   `-CallExpr 0x557c84bcf630 <col:14, col:31> 'int'
    |     |-ImplicitCastExpr 0x557c84bcf618 <col:14> 'int (*)(uint8_t)' <FunctionToPointerDecay>
    |     | `-DeclRefExpr 0x557c84bcf598 <col:14> 'int (uint8_t)' lvalue Function 0x557c84b4d320 'digitalRead' 'int (uint8_t)'
    |     `-ImplicitCastExpr 0x557c84bcf670 <col:26> 'uint8_t':'unsigned char' <IntegralCast>
    |       `-ImplicitCastExpr 0x557c84bcf658 <col:26> 'int' <LValueToRValue>
    |         `-DeclRefExpr 0x557c84bcf578 <col:26> 'int' lvalue Var 0x557c84bcee60 'inPin' 'int'
    |-CallExpr 0x557c84bcf7c0 <line:13:5, col:26> 'void'
    | |-ImplicitCastExpr 0x557c84bcf7a8 <col:5> 'void (*)(uint8_t, uint8_t)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x557c84bcf728 <col:5> 'void (uint8_t, uint8_t)' lvalue Function 0x557c84b4d160 'digitalWrite' 'void (uint8_t, uint8_t)'
    | |-ImplicitCastExpr 0x557c84bcf7f0 <col:18> 'uint8_t':'unsigned char' <IntegralCast>
    | | `-IntegerLiteral 0x557c84bcf6e8 <col:18> 'int' 10
    | `-ImplicitCastExpr 0x557c84bcf808 <./Arduino.h:40:14> 'uint8_t':'unsigned char' <IntegralCast>
    |   `-IntegerLiteral 0x557c84bcf708 <col:14> 'int' 1
    |-CallExpr 0x557c84bcf8e0 <Exsetup.cpp:14:5, col:23> 'void'
    | |-ImplicitCastExpr 0x557c84bcf8c8 <col:5> 'void (*)(uint8_t, uint8_t)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x557c84bcf8a8 <col:5> 'void (uint8_t, uint8_t)' lvalue Function 0x557c84b4cf90 'pinMode' 'void (uint8_t, uint8_t)'
    | |-ImplicitCastExpr 0x557c84bcf910 <col:14> 'uint8_t':'unsigned char' <IntegralCast>
    | | `-IntegerLiteral 0x557c84bcf868 <col:14> 'int' 13
    | `-ImplicitCastExpr 0x557c84bcf940 <col:18> 'uint8_t':'unsigned char' <IntegralCast>
    |   `-ImplicitCastExpr 0x557c84bcf928 <col:18> 'int' <LValueToRValue>
    |     `-DeclRefExpr 0x557c84bcf888 <col:18> 'int' lvalue Var 0x557c84b131a8 'INPUT' 'int'
    |-CallExpr 0x557c84bcfa50 <line:15:5, col:14> 'boolean':'bool'
    | |-ImplicitCastExpr 0x557c84bcfa38 <col:5> 'boolean (*)(int)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x557c84bcf9c0 <col:5> 'boolean (int)' lvalue Function 0x557c84b61ac0 'isAscii' 'boolean (int)'
    | `-ImplicitCastExpr 0x557c84bcfa90 <col:13> 'int' <IntegralCast>
    |   `-ImplicitCastExpr 0x557c84bcfa78 <col:13> 'char' <LValueToRValue>
    |     `-DeclRefExpr 0x557c84bcf9a0 <col:13> 'char' lvalue Var 0x557c84bceef8 'c' 'char'
    |-DeclStmt 0x557c84bd0110 <line:16:5, col:25>
    | `-VarDecl 0x557c84bcfac0 <col:5, col:24> col:11 x 'float' cinit
    |   `-ImplicitCastExpr 0x557c84bd00f8 <col:15, col:24> 'float' <FloatingCast>
    |     `-CallExpr 0x557c84bd00d0 <col:15, col:24> 'double'
    |       |-ImplicitCastExpr 0x557c84bd00b8 <col:15> 'double (*)(double) throw()' <FunctionToPointerDecay>
    |       | `-DeclRefExpr 0x557c84bd0060 <col:15> 'double (double) throw()' lvalue Function 0x557c848f6730 'sqrt' 'double (double) throw()' (UsingShadow 0x557c84ab0950 'sqrt')
    |       `-FloatingLiteral 0x557c84bcfb88 <col:20> 'double' 2.560000e+01
    `-CallExpr 0x557c84bd01f0 <line:17:5, col:14> 'void'
      |-ImplicitCastExpr 0x557c84bd01d8 <col:5> 'void (*)(unsigned long)' <FunctionToPointerDecay>
      | `-DeclRefExpr 0x557c84bd0190 <col:5> 'void (unsigned long)' lvalue Function 0x557c84b4dc78 'delay' 'void (unsigned long)'
      `-ImplicitCastExpr 0x557c84bd0218 <col:11> 'unsigned long' <IntegralCast>
        `-IntegerLiteral 0x557c84bd0170 <col:11> 'int' 100
9 warnings and 1 error generated.
ashutosh@omen:~/clang-llvm/llvm-project/Arduino-headerfiles$
```

Then we write a FrontEndAction along with a RecursiveASTVisitor to visit our nodes recursively. We also define a ASTConsumer interface. There is some boilerplate code that has to be only defined once:

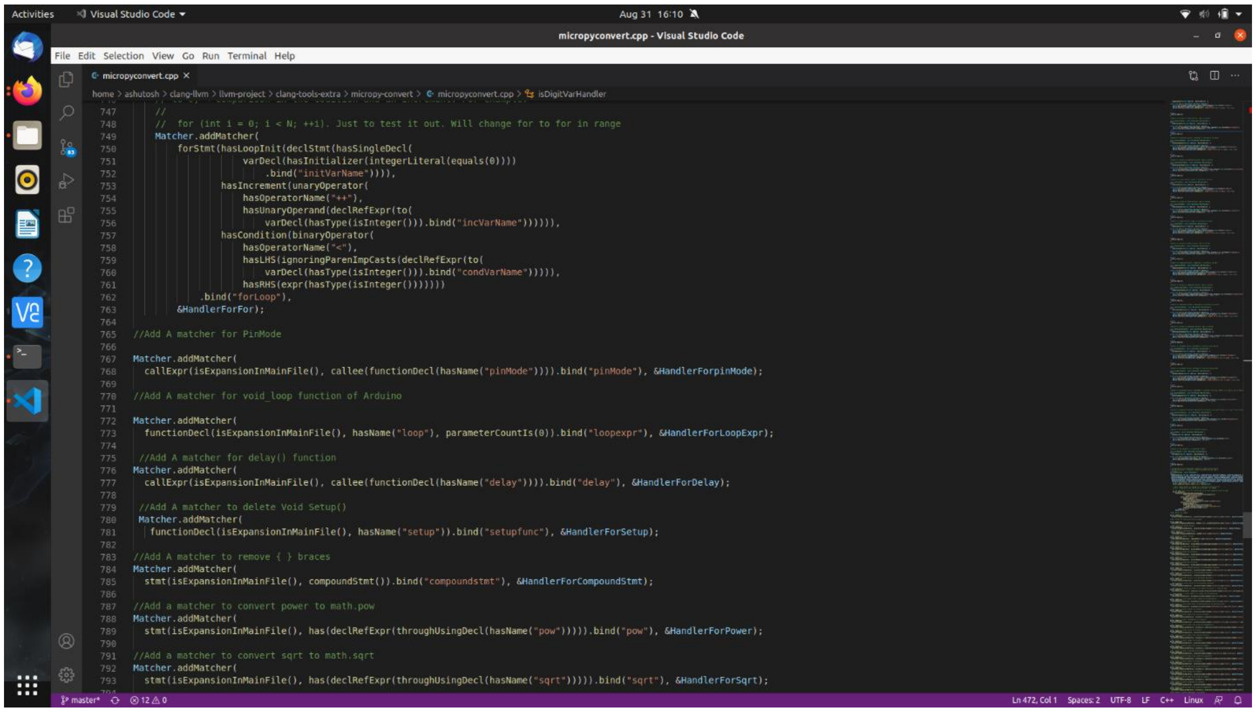


```cpp
  eulerHandler HandlerForEuler;


  MatchFinder Matcher;
};

// For each source file provided to the tool, a new FrontendAction is created.
class MyFrontendAction : public ASTFrontendAction {
public:
  MyFrontendAction() {}
  void EndSourceFileAction() override {
   SourceManager &SM = TheRewriter.getSourceMgr();
   llvm::errs() << "** EndSourceFileAction for: "
                << SM.getFileEntryForID(SM.getMainFileID())->getName() << "\n";
//Now emit the Rewritten Buffer
    TheRewriter.getEditBuffer(TheRewriter.getSourceMgr().getMainFileID())
        .write(llvm::outs());

std::error_code error_code;
        llvm::raw_fd_ostream outFile("output.txt", error_code, llvm::sys::fs::F_None);
     TheRewriter.getEditBuffer(SM.getMainFileID()).write(outFile); // --> this will write the result>
    outFile.close();

  }

  std::unique_ptr<ASTConsumer> CreateASTConsumer(CompilerInstance &CI,
                                                 StringRef file) override {
    TheRewriter.setSourceMgr(CI.getSourceManager(), CI.getLangOpts());
    return std::make_unique<MyASTConsumer>(TheRewriter);
  }

private:
  Rewriter TheRewriter;
};

int main(int argc, const char **argv) {
  CommonOptionsParser op(argc, argv, MatcherSampleCategory);
  ClangTool Tool(op.getCompilations(), op.getSourcePathList());

  return Tool.run(newFrontendActionFactory<MyFrontendAction>().get());
}
```

Then we identify a function, statement, or variable that we want to alter. We see its representation in the AST, and use clang query along with the AST matcher Reference. Using this reference and our knowledge of the AST, we write custom matchers with patterns (and sometimes, antipatterns) to narrow our query down to a specific node. Most of the functions are defined as CallExpr or DeclRefExpr and can be traced to their declarations in the headers via Callee() and hasAncestor() functions. Other standard C style expressions such as isAlpha(), isAscii() are recorded as ShadowDecl nodes, so we use that. The matchers are then binded with a keyword.



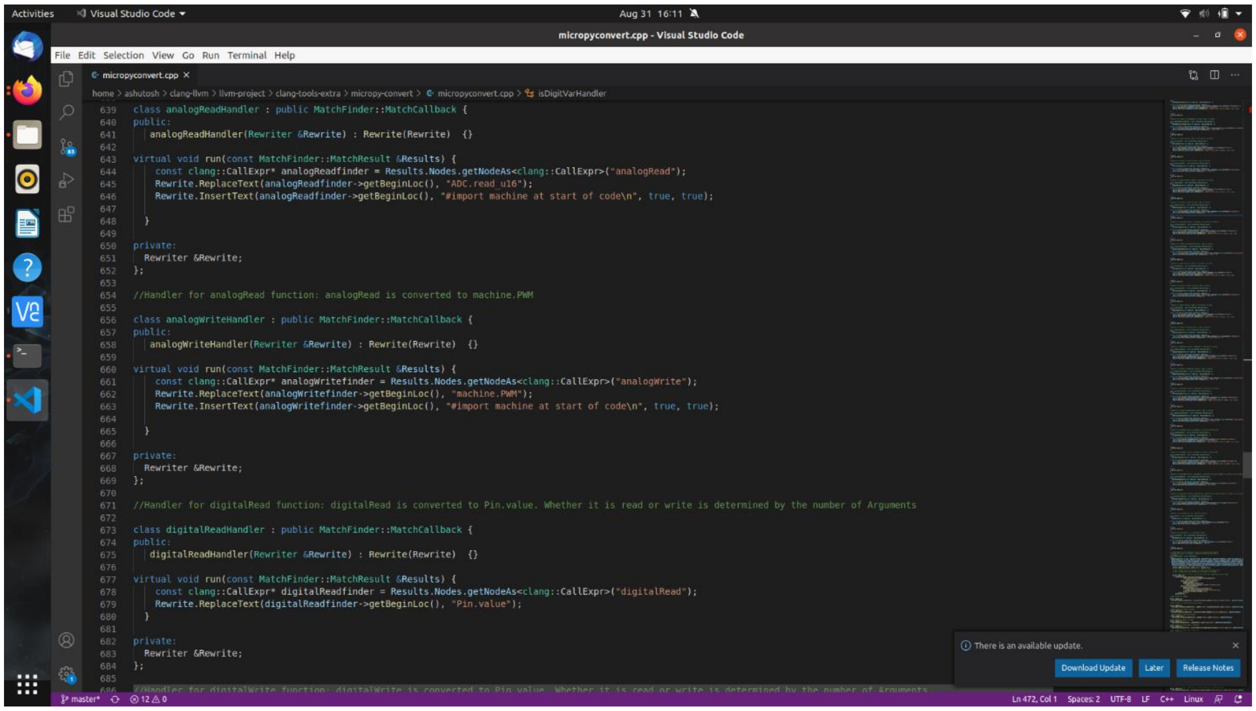
```
clang-query> m callExpr(isExpansionInMainFile(), callee(functionDecl(hasName("pinMode")))).bind("pinMode")

Match #1:

/home/ashutosh/clang-llvm/llvm-project/Arduino-headerfiles/Exsetup.cpp:14:5: note: "pinMode" binds here
    pinMode( 13, INPUT);
    ^~~~~~~~~~~~~~~~~~~
/home/ashutosh/clang-llvm/llvm-project/Arduino-headerfiles/Exsetup.cpp:14:5: note: "root" binds here
    pinMode( 13, INPUT);
    ^~~~~~~~~~~~~~~~~~~
1 match.
clang-query> 
```



Activities
Visual Studio Code
Aug 31 16:10
micropyconvert.cpp - Visual Studio Code
File Edit Selection View Go Run Terminal Help
micropyconvert.cpp
home > ashutosh > clang-llvm > llvm-project > clang-tools-extra > micropy-convert > micropyconvert.cpp > isDigitVarHandler
master* ⊗ 12 ⚠ 0
Ln 472, Col 1 Spaces: 2 UTF-8 LF C++ Linux

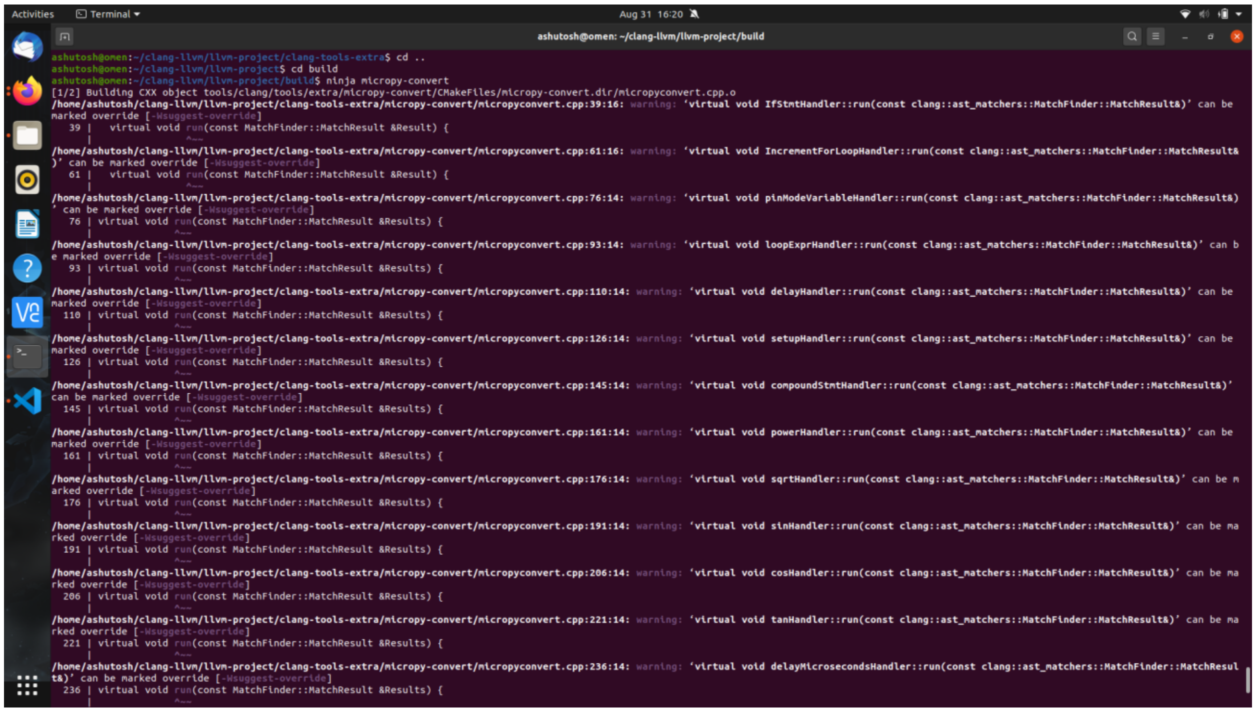
```
    //
    //  for (int i = 0; i < N; ++i). Just to test it out. Will change for to for in range
    Matcher.addMatcher(
        forStmt(hasLoopInit(declStmt(hasSingleDecl(
                    varDecl(hasInitializer(integerLiteral(equals(0))))
                        .bind("initVarName")))),
                hasIncrement(unaryOperator(
                    hasOperatorName("++"),
                    hasUnaryOperand(declRefExpr(to(
                        varDecl(hasType(isInteger())).bind("incVarName")))))),
                hasCondition(binaryOperator(
                    hasOperatorName("<"),
                    hasLHS(ignoringParenImpCasts(declRefExpr(to(
                        varDecl(hasType(isInteger())).bind("condVarName"))))),
                    hasRHS(expr(hasType(isInteger()))))))
            .bind("forLoop"),
        &HandlerForFor);

  //Add A matcher for PinMode

  Matcher.addMatcher(
    callExpr(isExpansionInMainFile(), callee(functionDecl(hasName("pinMode")))).bind("pinMode"), &HandlerForpinMode);

  //Add A matcher for void_loop function of Arduino

  Matcher.addMatcher(
    functionDecl(isExpansionInMainFile(), hasName("loop"), parameterCountIs(0)).bind("loopexpr"), &HandlerForLoopExpr);

   //Add A matcher for delay() function
  Matcher.addMatcher(
    callExpr(isExpansionInMainFile(), callee(functionDecl(hasName("delay")))).bind("delay"), &HandlerForDelay);

   //Add A matcher to delete Void Setup()
   Matcher.addMatcher(
     functionDecl(isExpansionInMainFile(), hasName("setup")).bind("setupfunc"), &HandlerForSetup);

  //Add A matcher to remove { } braces
  Matcher.addMatcher(
    stmt(isExpansionInMainFile(), compoundStmt()).bind("compoundstmt"), &HandlerForCompoundStmt);

  //Add a matcher to convert power to math.pow
  Matcher.addMatcher(
    stmt(isExpansionInMainFile(), has(declRefExpr(throughUsingDecl(hasName("pow"))))).bind("pow"), &HandlerForPower);

  //Add a matcher to convert sqrt to math.sqrt
  Matcher.addMatcher(
    stmt(isExpansionInMainFile(), has(declRefExpr(throughUsingDecl(hasName("sqrt"))))).bind("sqrt"), &HandlerForSqrt);
```

Next, we define a Handler Class for each node. Again, there is some boilerplate code which is repeated every time we want to access a node. We use the rewrite class to make precise edits on the AST, transforming them to the equivalent MicroPython expressions. We can also include warnings for transformations that need specific modules to work.



```
class analogReadHandler : public MatchFinder::MatchCallback {
public:
  analogReadHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* analogReadfinder = Results.Nodes.getNodeAs<clang::CallExpr>("analogRead");
    Rewrite.ReplaceText(analogReadfinder->getBeginLoc(), "ADC.read_u16");
    Rewrite.InsertText(analogReadfinder->getBeginLoc(), "#import machine at start of code\n", true, true);

  }

private:
  Rewriter &Rewrite;
};

//Handler for analogRead function: analogRead is converted to machine.PWM

class analogWriteHandler : public MatchFinder::MatchCallback {
public:
  analogWriteHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* analogWritefinder = Results.Nodes.getNodeAs<clang::CallExpr>("analogWrite");
    Rewrite.ReplaceText(analogWritefinder->getBeginLoc(), "machine.PWM");
    Rewrite.InsertText(analogWritefinder->getBeginLoc(), "#import machine at start of code\n", true, true);

  }

private:
  Rewriter &Rewrite;
};

//Handler for digitalRead function: digitalRead is converted to Pin.value. Whether it is read or write is determined by the number of Arguments

class digitalReadHandler : public MatchFinder::MatchCallback {
public:
  digitalReadHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* digitalReadfinder = Results.Nodes.getNodeAs<clang::CallExpr>("digitalRead");
    Rewrite.ReplaceText(digitalReadfinder->getBeginLoc(), "Pin.value");
  }

private:
  Rewriter &Rewrite;
};

//Handler for digitalWrite function: digitalWrite is converted to Pin.value. Whether it is read or write is determined by the number of Arguments
```

We rebuild the tool after saving it, using the ninja keyword in the build directory:



```
ashutosh@omen:~/clang-llvm/llvm-project/clang-tools-extra$ cd ..
ashutosh@omen:~/clang-llvm/llvm-project$ cd build
ashutosh@omen:~/clang-llvm/llvm-project/build$ ninja micropy-convert
[1/2] Building CXX object tools/clang/tools/extra/micropy-convert/CMakeFiles/micropy-convert.dir/micropyconvert.cpp.o
/home/ashutosh/clang-llvm/llvm-project/clang-tools-extra/micropy-convert/micropyconvert.cpp:39:16: warning: 'virtual void IfStmtHandler::run(const clang::ast_matchers::MatchFinder::MatchResult&)' can be
marked override [-Wsuggest-override]
   39 |   virtual void run(const MatchFinder::MatchResult &Result) {
      |                ^~~
/home/ashutosh/clang-llvm/llvm-project/clang-tools-extra/micropy-convert/micropyconvert.cpp:61:16: warning: 'virtual void IncrementForLoopHandler::run(const clang::ast_matchers::MatchFinder::MatchResult&
)' can be marked override [-Wsuggest-override]
   61 |   virtual void run(const MatchFinder::MatchResult &Result) {
      |                ^~~
/home/ashutosh/clang-llvm/llvm-project/clang-tools-extra/micropy-convert/micropyconvert.cpp:76:14: warning: 'virtual void pinModeVariableHandler::run(const clang::ast_matchers::MatchFinder::MatchResult&)
' can be marked override [-Wsuggest-override]
   76 | virtual void run(const MatchFinder::MatchResult &Results) {
      |              ^~~
/home/ashutosh/clang-llvm/llvm-project/clang-tools-extra/micropy-convert/micropyconvert.cpp:93:14: warning: 'virtual void loopExprHandler::run(const clang::ast_matchers::MatchFinder::MatchResult&)' can b
e marked override [-Wsuggest-override]
   93 | virtual void run(const MatchFinder::MatchResult &Results) {
      |              ^~~
/home/ashutosh/clang-llvm/llvm-project/clang-tools-extra/micropy-convert/micropyconvert.cpp:110:14: warning: 'virtual void delayHandler::run(const clang::ast_matchers::MatchFinder::MatchResult&)' can be
marked override [-Wsuggest-override]
  110 | virtual void run(const MatchFinder::MatchResult &Results) {
      |              ^~~
/home/ashutosh/clang-llvm/llvm-project/clang-tools-extra/micropy-convert/micropyconvert.cpp:126:14: warning: 'virtual void setupHandler::run(const clang::ast_matchers::MatchFinder::MatchResult&)' can be
marked override [-Wsuggest-override]
  126 | virtual void run(const MatchFinder::MatchResult &Results) {
      |              ^~~
/home/ashutosh/clang-llvm/llvm-project/clang-tools-extra/micropy-convert/micropyconvert.cpp:145:14: warning: 'virtual void compoundStmtHandler::run(const clang::ast_matchers::MatchFinder::MatchResult&)'
can be marked override [-Wsuggest-override]
  145 | virtual void run(const MatchFinder::MatchResult &Results) {
      |              ^~~
/home/ashutosh/clang-llvm/llvm-project/clang-tools-extra/micropy-convert/micropyconvert.cpp:161:14: warning: 'virtual void powerHandler::run(const clang::ast_matchers::MatchFinder::MatchResult&)' can be
marked override [-Wsuggest-override]
  161 | virtual void run(const MatchFinder::MatchResult &Results) {
      |              ^~~
/home/ashutosh/clang-llvm/llvm-project/clang-tools-extra/micropy-convert/micropyconvert.cpp:176:14: warning: 'virtual void sqrtHandler::run(const clang::ast_matchers::MatchFinder::MatchResult&)' can be m
arked override [-Wsuggest-override]
  176 | virtual void run(const MatchFinder::MatchResult &Results) {
      |              ^~~
/home/ashutosh/clang-llvm/llvm-project/clang-tools-extra/micropy-convert/micropyconvert.cpp:191:14: warning: 'virtual void sinHandler::run(const clang::ast_matchers::MatchFinder::MatchResult&)' can be ma
rked override [-Wsuggest-override]
  191 | virtual void run(const MatchFinder::MatchResult &Results) {
      |              ^~~
/home/ashutosh/clang-llvm/llvm-project/clang-tools-extra/micropy-convert/micropyconvert.cpp:206:14: warning: 'virtual void cosHandler::run(const clang::ast_matchers::MatchFinder::MatchResult&)' can be ma
rked override [-Wsuggest-override]
  206 | virtual void run(const MatchFinder::MatchResult &Results) {
      |              ^~~
/home/ashutosh/clang-llvm/llvm-project/clang-tools-extra/micropy-convert/micropyconvert.cpp:221:14: warning: 'virtual void tanHandler::run(const clang::ast_matchers::MatchFinder::MatchResult&)' can be ma
rked override [-Wsuggest-override]
  221 | virtual void run(const MatchFinder::MatchResult &Results) {
      |              ^~~
/home/ashutosh/clang-llvm/llvm-project/clang-tools-extra/micropy-convert/micropyconvert.cpp:236:14: warning: 'virtual void delayMicrosecondsHandler::run(const clang::ast_matchers::MatchFinder::MatchResul
t&)' can be marked override [-Wsuggest-override]
  236 | virtual void run(const MatchFinder::MatchResult &Results) {
      |              ^~~
```

Using this process, almost any part of the code can be accessed and edited.

Advantages of this Approach:

- The nodes can be made very specific, so syntactically incorrect code will not be converted.
- The same technique can be used to develop a CPP checker.
- Key words only in a specific node are edited, leaving out the rest.
- Code that is not transformable is left as is, so that the user can edit it out.
- Can include error and diagnostic messages.

Future Improvements:

- Currently the Serial class is broken and cannot be accessed by the Clang AST, this needs to be fixed by editing the headers and dependencies. Clang has AVR support but its not full fledged.
- Arduino's MicroPython port is in its infancy, and as hardware specific implementations become more clear, the tool will develop to accomodate that.

Code

CMAKELISTS.txt

```
set(LLVM_LINK_COMPONENTS support)

add_clang_executable(micropy-convert
        micropyconvert.cpp
        )
target_link_libraries(micropy-convert
        PRIVATE
        clangTooling
        clangBasic
        clangASTMatchers
        )
```

Clang/LLVM

```
//----------------------------------------------------------------------------
---
// Micropython-Convert Demonstrates:
//
// * How to convert Arduino Sketches to Micropython
// * How to use AST matchers to find interesting AST nodes.
// * How to use the Rewriter API to rewrite the source code.
//
// Ashutosh Pandey (ashutoshpandey123456@gmail.com)
// This code is in the public domain
//----------------------------------------------------------------------------
---
#include <string>
```

```
#include "clang/AST/AST.h"
#include "clang/AST/ASTConsumer.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendActions.h"
#include "clang/Rewrite/Core/Rewriter.h"
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/raw_ostream.h"
#include "clang/AST/Expr.h"

using namespace std;
using namespace clang;
using namespace clang::ast_matchers;
using namespace clang::driver;
using namespace clang::tooling;

static llvm::cl::OptionCategory MatcherSampleCategory("Matcher Sample");

//IfStatementHandler Class: All Rewriting For IF statements done here.

class IfStmtHandler : public MatchFinder::MatchCallback {
public:
  IfStmtHandler(Rewriter &Rewrite) : Rewrite(Rewrite) {}

  virtual void run(const MatchFinder::MatchResult &Result) {
    // The matched 'if' statement was bound to 'ifStmt'.
    if (const IfStmt *IfS = Result.Nodes.getNodeAs<clang::IfStmt>("ifStmt"))
{
      const Stmt *Then = IfS->getThen();
      Rewrite.InsertText(Then->getBeginLoc(), "#if part\n", true, true);

      if (const Stmt *Else = IfS->getElse()) {
        Rewrite.InsertText(Else->getBeginLoc(), "#else part\n", true, true);
      }
    }
  }

private:
  Rewriter &Rewrite;
};

//ForLoopHandler Class: All Rewriting For For Loop statements done here.

class IncrementForLoopHandler : public MatchFinder::MatchCallback {
public:
  IncrementForLoopHandler(Rewriter &Rewrite) : Rewrite(Rewrite) {}

  virtual void run(const MatchFinder::MatchResult &Result) {
    const VarDecl *IncVar = Result.Nodes.getNodeAs<VarDecl>("incVarName");
    Rewrite.InsertText(IncVar->getBeginLoc(), "#incvar/n", true, true);
  }

private:
  Rewriter &Rewrite;
};

//PinMode Class: All Rewriting For PinMode statements done here.

class pinModeVariableHandler : public MatchFinder::MatchCallback {
public:
   pinModeVariableHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* pm =
Results.Nodes.getNodeAs<clang::CallExpr>("pinMode");
    Rewrite.ReplaceText(pm->getBeginLoc(), "Pin.mode");
    Rewrite.InsertText(pm->getBeginLoc(), "#from machine import pin at start
of code\n", true, true);

  }

private:
  Rewriter &Rewrite;
};

//Handler for Void Loop() Class: All Rewriting For void loop statements done
here. Void loop() is rewritten as While True:

class loopExprHandler : public MatchFinder::MatchCallback {
public:
   loopExprHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::FunctionDecl* loop =
Results.Nodes.getNodeAs<clang::FunctionDecl>("loopexpr");
    Rewrite.RemoveText(loop->getLocation());
    Rewrite.ReplaceText(loop->getBeginLoc(), "While True:");
    Rewrite.ReplaceText(loop->getLocation(), " ");
  }

private:
  Rewriter &Rewrite;
};

//Handler for delay() function: delay() is rewritten as time.sleep_ms

class delayHandler : public MatchFinder::MatchCallback {
public:
   delayHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* delayfinder =
Results.Nodes.getNodeAs<clang::CallExpr>("delay");
    Rewrite.ReplaceText(delayfinder->getBeginLoc(), "utime.sleep_ms");

  }

private:
  Rewriter &Rewrite;
};
```

```
//Handler for Void Setup() Class: Void Setup is Deleted as It does not occur
in Micropython Statements

class setupHandler : public MatchFinder::MatchCallback {
public:
   setupHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::FunctionDecl* setupfinder =
Results.Nodes.getNodeAs<clang::FunctionDecl>("setupfunc");
    Rewrite.RemoveText(setupfinder->getLocation());
    Rewrite.RemoveText(setupfinder->getBeginLoc());
    Rewrite.ReplaceText(setupfinder->getBeginLoc(), " ");
  }

private:
  Rewriter &Rewrite;
};


//Handler for CompoundStatements: Curly Braces are not required in
Micropython and Can be removed. Since  project with improper indentation
might become hard to understand
// it will insert a # (comment statement) before each {}

class compoundStmtHandler : public MatchFinder::MatchCallback {
public:
   compoundStmtHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::Stmt* compoundstmtfinder =
Results.Nodes.getNodeAs<clang::Stmt>("compoundstmt");
    Rewrite.InsertText(compoundstmtfinder->getBeginLoc(), "#", true, true);
    Rewrite.InsertText(compoundstmtfinder->getEndLoc(), "#", true, true);
  }

private:
  Rewriter &Rewrite;
};

//Handler for power expression. converts pow to math.pow

class powerHandler : public MatchFinder::MatchCallback {
public:
   powerHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::Stmt* powfinder =
Results.Nodes.getNodeAs<clang::Stmt>("pow");
    Rewrite.InsertText(powfinder->getBeginLoc(), "math.", true, true);
  }

private:
  Rewriter &Rewrite;
};

//Handler for square root expression. converts sqrt to math.sqrt
```

```
class sqrtHandler : public MatchFinder::MatchCallback {
public:
   sqrtHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::Stmt* sqrtfinder =
Results.Nodes.getNodeAs<clang::Stmt>("sqrt");
    Rewrite.InsertText(sqrtfinder->getBeginLoc(), "math.", true, true);
  }

private:
  Rewriter &Rewrite;
};

//Handler for sin expression. converts sin to math.sin

class sinHandler : public MatchFinder::MatchCallback {
public:
   sinHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::Stmt* sinfinder =
Results.Nodes.getNodeAs<clang::Stmt>("sin");
    Rewrite.InsertText(sinfinder->getBeginLoc(), "math.", true, true);
  }

private:
  Rewriter &Rewrite;
};

//Handler for cos expression. converts cos to math.cos

class cosHandler : public MatchFinder::MatchCallback {
public:
   cosHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::Stmt* cosfinder =
Results.Nodes.getNodeAs<clang::Stmt>("cos");
    Rewrite.InsertText(cosfinder->getBeginLoc(), "math.", true, true);
  }

private:
  Rewriter &Rewrite;
};

//Handler for tan expression. converts tan to math.tan

class tanHandler : public MatchFinder::MatchCallback {
public:
   tanHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::Stmt* tanfinder =
Results.Nodes.getNodeAs<clang::Stmt>("tan");
    Rewrite.InsertText(tanfinder->getBeginLoc(), "math.", true, true);
  }

private:
  Rewriter &Rewrite;
};

//Handler for delay() function: delay() is rewritten as time.sleep_ms

class delayMicrosecondsHandler : public MatchFinder::MatchCallback {
public:
   delayMicrosecondsHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* delayMicrosecondsfinder =
Results.Nodes.getNodeAs<clang::CallExpr>("delayMicroseconds");
    Rewrite.ReplaceText(delayMicrosecondsfinder->getBeginLoc(),
"utime.sleep_us");
  }

private:
  Rewriter &Rewrite;
};

//Handler for delay() function: delay() is rewritten as time.sleep_ms

class millisHandler : public MatchFinder::MatchCallback {
public:
   millisHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* millisfinder =
Results.Nodes.getNodeAs<clang::CallExpr>("millis");
    Rewrite.ReplaceText(millisfinder->getBeginLoc(), "utime.ticks_ms");
  }

private:
  Rewriter &Rewrite;
};

//Handler for delay() function: delay() is rewritten as time.sleep_ms

class microsHandler : public MatchFinder::MatchCallback {
public:
   microsHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* microsfinder =
Results.Nodes.getNodeAs<clang::CallExpr>("micros");
    Rewrite.ReplaceText(microsfinder->getBeginLoc(), "utime.ticks_us");
  }

private:
  Rewriter &Rewrite;
};

//Handler for delay() function: delay() is rewritten as time.sleep_ms
```

```
class pulseInHandler : public MatchFinder::MatchCallback {
public:
   pulseInHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* pulseInfinder =
Results.Nodes.getNodeAs<clang::CallExpr>("pulseIn");
    Rewrite.ReplaceText(pulseInfinder->getBeginLoc(),
"machine.time_pulse_us");
  }

private:
  Rewriter &Rewrite;
};

//Handler for PinMode Pin. converts pin number  to p<pinNumber>

class pinModePinHandler : public MatchFinder::MatchCallback {
public:
   pinModePinHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::Stmt* pinModePinfinder =
Results.Nodes.getNodeAs<clang::Stmt>("pinModePin");
    Rewrite.InsertText(pinModePinfinder->getBeginLoc(), "p", true, true);
  }

private:
  Rewriter &Rewrite;
};
//Handler for INPUT keyword converts to IN

class inputHandler : public MatchFinder::MatchCallback {
public:
   inputHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::Stmt* inputfinder =
Results.Nodes.getNodeAs<clang::Stmt>("INPUT");
        Rewrite.ReplaceText(inputfinder->getBeginLoc(), "IN");
  }

private:
  Rewriter &Rewrite;
};
 //Handler for OUTPUT keyword converts to OUT

class outputHandler : public MatchFinder::MatchCallback {
public:
   outputHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::Stmt* outputfinder =
Results.Nodes.getNodeAs<clang::Stmt>("OUTPUT");
    Rewrite.ReplaceText(outputfinder->getBeginLoc(), "OUT");

  }
private:
  Rewriter &Rewrite;
};
//Handler for INPUT_PULLUP keyword converts to PULL_UP

class inputpullupHandler : public MatchFinder::MatchCallback {
public:
   inputpullupHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::Stmt* inputpullupfinder =
Results.Nodes.getNodeAs<clang::Stmt>("INPUT_PULLUP");
    Rewrite.ReplaceText(inputpullupfinder->getBeginLoc(), "PULL_UP");
  }

private:
  Rewriter &Rewrite;

};

//Handler for isAlpha function: rewritten as ure.match()

class isAlphaHandler : public MatchFinder::MatchCallback {
public:
   isAlphaHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* isAlphafinder =
Results.Nodes.getNodeAs<clang::CallExpr>("isAlpha");
    Rewrite.ReplaceText(isAlphafinder->getBeginLoc(), "ure.match");
    Rewrite.InsertText(isAlphafinder->getBeginLoc(), "#import ure at start of
code\n", true, true);

  }

private:
  Rewriter &Rewrite;
};

//Handler for variable in isAlpha function: regex is inserted

class isAlphaVarHandler : public MatchFinder::MatchCallback {
public:
   isAlphaVarHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::DeclRefExpr* isAlphaVarfinder =
Results.Nodes.getNodeAs<clang::DeclRefExpr>("isAlphaVar");
    Rewrite.InsertText(isAlphaVarfinder->getBeginLoc(), "'[A-Za-z]', ");
  }

private:
  Rewriter &Rewrite;
};

//Handler for isAlphaNumeric function: rewritten as ure.match()
```

```
class isAlphaNumericHandler : public MatchFinder::MatchCallback {
public:
   isAlphaNumericHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* isAlphaNumericfinder =
Results.Nodes.getNodeAs<clang::CallExpr>("isAlphaNumeric");
    Rewrite.ReplaceText(isAlphaNumericfinder->getBeginLoc(), "ure.match");
    Rewrite.InsertText(isAlphaNumericfinder->getBeginLoc(), "#import ure at
start of code\n", true, true);

  }

private:
  Rewriter &Rewrite;
};

//Handler for variable in isAlphaNumeric function: regex is inserted

class isAlphaNumericVarHandler : public MatchFinder::MatchCallback {
public:
   isAlphaNumericVarHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::DeclRefExpr* isAlphaNumericVarfinder =
Results.Nodes.getNodeAs<clang::DeclRefExpr>("isAlphaNumericVar");
    Rewrite.InsertText(isAlphaNumericVarfinder->getBeginLoc(), "'[A-Za-z0-
9]', ");
  }

private:
  Rewriter &Rewrite;
};

//Handler for isAscii function: isAscii is rewritten as ure.match()

class isAsciiHandler : public MatchFinder::MatchCallback {
public:
   isAsciiHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* isAsciifinder =
Results.Nodes.getNodeAs<clang::CallExpr>("isAscii");
    Rewrite.ReplaceText(isAsciifinder->getBeginLoc(), "ure.match");
    Rewrite.InsertText(isAsciifinder->getBeginLoc(), "#import ure at start of
code\n", true, true);

  }

private:
  Rewriter &Rewrite;
};

//Handler for variable in isAscii function: regex is inserted.

class isAsciiVarHandler : public MatchFinder::MatchCallback {
```

```
public:
   isAsciiVarHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::DeclRefExpr* isAsciiVarfinder =
Results.Nodes.getNodeAs<clang::DeclRefExpr>("isAsciiVar");
    Rewrite.InsertText(isAsciiVarfinder->getBeginLoc(), "'\\w\\W' ");
  }

private:
  Rewriter &Rewrite;
};

//Handler for isDigit function: isDigit is rewritten as ure.match()

class isDigitHandler : public MatchFinder::MatchCallback {
public:
   isDigitHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* isDigitfinder =
Results.Nodes.getNodeAs<clang::CallExpr>("isDigit");
    Rewrite.ReplaceText(isDigitfinder->getBeginLoc(), "ure.match");
    Rewrite.InsertText(isDigitfinder->getBeginLoc(), "#import ure at start of
code\n", true, true);

  }

private:
  Rewriter &Rewrite;
};

//Handler for variable in isDigit function: regex is inserted.

class isDigitVarHandler : public MatchFinder::MatchCallback {
public:
   isDigitVarHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::DeclRefExpr* isDigitVarfinder =
Results.Nodes.getNodeAs<clang::DeclRefExpr>("isDigitVar");
    Rewrite.InsertText(isDigitVarfinder->getBeginLoc(), "'\\d' ");
  }

private:
  Rewriter &Rewrite;
};

//Handler for isLowerCase function: isLowerCase is rewritten as ure.match()

class isLowerCaseHandler : public MatchFinder::MatchCallback {
public:
   isLowerCaseHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* isLowerCasefinder =
Results.Nodes.getNodeAs<clang::CallExpr>("isLowerCase");
```

```
    Rewrite.ReplaceText(isLowerCasefinder->getBeginLoc(), "ure.match");
    Rewrite.InsertText(isLowerCasefinder->getBeginLoc(), "#import ure at
start of code\n", true, true);

  }

private:
  Rewriter &Rewrite;
};

//Handler for variable in isLowerCase function: regex is inserted.

class isLowerCaseVarHandler : public MatchFinder::MatchCallback {
public:
   isLowerCaseVarHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::DeclRefExpr* isLowerCaseVarfinder =
Results.Nodes.getNodeAs<clang::DeclRefExpr>("isLowerCaseVar");
    Rewrite.InsertText(isLowerCaseVarfinder->getBeginLoc(), "'[a-z]', ");
  }

private:
  Rewriter &Rewrite;
};

//Handler for isPunct function: isPunct is rewritten as ure.match

class isPunctHandler : public MatchFinder::MatchCallback {
public:
   isPunctHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* isPunctfinder =
Results.Nodes.getNodeAs<clang::CallExpr>("isPunct");
    Rewrite.ReplaceText(isPunctfinder->getBeginLoc(), "ure.match");
    Rewrite.InsertText(isPunctfinder->getBeginLoc(), "#import ure at start of
code\n", true, true);

  }

private:
  Rewriter &Rewrite;
};

//Handler for variable in isPunct function: regex is inserted.

class isPunctVarHandler : public MatchFinder::MatchCallback {
public:
   isPunctVarHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::DeclRefExpr* isPunctVarfinder =
Results.Nodes.getNodeAs<clang::DeclRefExpr>("isPunctVar");
    Rewrite.InsertText(isPunctVarfinder->getBeginLoc(), "'\\W' ");
  }
```

```
private:
  Rewriter &Rewrite;
};

//Handler for isSpace function: isSpace is rewritten as ure.match

class isSpaceHandler : public MatchFinder::MatchCallback {
public:
   isSpaceHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* isSpacefinder =
Results.Nodes.getNodeAs<clang::CallExpr>("isSpace");
    Rewrite.ReplaceText(isSpacefinder->getBeginLoc(), "ure.match");
    Rewrite.InsertText(isSpacefinder->getBeginLoc(), "#import ure at start of
code\n", true, true);

  }

private:
  Rewriter &Rewrite;
};

//Handler for variable in isSpace function: regex is inserted.

class isSpaceVarHandler : public MatchFinder::MatchCallback {
public:
   isSpaceVarHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::DeclRefExpr* isSpaceVarfinder =
Results.Nodes.getNodeAs<clang::DeclRefExpr>("isSpaceVar");
    Rewrite.InsertText(isSpaceVarfinder->getBeginLoc(),
"'\\f\\n\\r\\t\\v\\s', ");
  }

private:
  Rewriter &Rewrite;
};

//Handler for isUpperCase function: isUpperCase is rewritten as ure.match

class isUpperCaseHandler : public MatchFinder::MatchCallback {
public:
   isUpperCaseHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* isUpperCasefinder =
Results.Nodes.getNodeAs<clang::CallExpr>("isUpperCase");
    Rewrite.ReplaceText(isUpperCasefinder->getBeginLoc(), "ure.match");
    Rewrite.InsertText(isUpperCasefinder->getBeginLoc(), "#import ure at
start of code\n", true, true);

  }

private:
  Rewriter &Rewrite;
```

```
};

//Handler for variable in isUpperCase function: regex is inserted.

class isUpperCaseVarHandler : public MatchFinder::MatchCallback {
public:
   isUpperCaseVarHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::DeclRefExpr* isUpperCaseVarfinder =
Results.Nodes.getNodeAs<clang::DeclRefExpr>("isUpperCaseVar");
    Rewrite.InsertText(isUpperCaseVarfinder->getBeginLoc(), "'[A-Z]', ");
  }

private:
  Rewriter &Rewrite;
};

//Handler for isWhitespace function: isWhitespace is rewritten as ure.match

class isWhitespaceHandler : public MatchFinder::MatchCallback {
public:
   isWhitespaceHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* isWhitespacefinder =
Results.Nodes.getNodeAs<clang::CallExpr>("isWhitespace");
    Rewrite.ReplaceText(isWhitespacefinder->getBeginLoc(), "ure.match");
    Rewrite.InsertText(isWhitespacefinder->getBeginLoc(), "#import ure at
start of code\n", true, true);

  }

private:
  Rewriter &Rewrite;
};

//Handler for variable in isWhitespace function: regex is inserted.

class isWhitespaceVarHandler : public MatchFinder::MatchCallback {
public:
   isWhitespaceVarHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::DeclRefExpr* isWhitespaceVarfinder =
Results.Nodes.getNodeAs<clang::DeclRefExpr>("isWhitespaceVar");
    Rewrite.InsertText(isWhitespaceVarfinder->getBeginLoc(), "'\\s\\t', ");
  }

private:
  Rewriter &Rewrite;
};

//Handler for analogRead function: analogRead is converted to ADC.read_u16

class analogReadHandler : public MatchFinder::MatchCallback {
public:
   analogReadHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* analogReadfinder =
Results.Nodes.getNodeAs<clang::CallExpr>("analogRead");
    Rewrite.ReplaceText(analogReadfinder->getBeginLoc(), "ADC.read_u16");
    Rewrite.InsertText(analogReadfinder->getBeginLoc(), "#import machine at
start of code\n", true, true);

  }

private:
  Rewriter &Rewrite;
};

//Handler for analogRead function: analogRead is converted to machine.PWM

class analogWriteHandler : public MatchFinder::MatchCallback {
public:
   analogWriteHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* analogWritefinder =
Results.Nodes.getNodeAs<clang::CallExpr>("analogWrite");
    Rewrite.ReplaceText(analogWritefinder->getBeginLoc(), "machine.PWM");
    Rewrite.InsertText(analogWritefinder->getBeginLoc(), "#import machine at
start of code\n", true, true);

  }

private:
  Rewriter &Rewrite;
};

//Handler for digitalRead function: digitalRead is converted to Pin.value.
Whether it is read or write is determined by the number of Arguments

class digitalReadHandler : public MatchFinder::MatchCallback {
public:
   digitalReadHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* digitalReadfinder =
Results.Nodes.getNodeAs<clang::CallExpr>("digitalRead");
    Rewrite.ReplaceText(digitalReadfinder->getBeginLoc(), "Pin.value");
  }

private:
  Rewriter &Rewrite;
};

//Handler for digitalWrite function: digitalWrite is converted to Pin.value.
Whether it is read or write is determined by the number of Arguments.

class digitalWriteHandler : public MatchFinder::MatchCallback {
public:
   digitalWriteHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}
```

```
virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::CallExpr* digitalWritefinder =
Results.Nodes.getNodeAs<clang::CallExpr>("digitalWrite");
    Rewrite.ReplaceText(digitalWritefinder->getBeginLoc(), "Pin.value");
  }

private:
  Rewriter &Rewrite;
};

//Handler for the constant Pi. Pi is converted to math.pi

class piHandler : public MatchFinder::MatchCallback {
public:
   piHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::DeclRefExpr* pifinder =
Results.Nodes.getNodeAs<clang::DeclRefExpr>("PI");
    Rewrite.ReplaceText(pifinder->getBeginLoc(), "math.pi");
  }

private:
  Rewriter &Rewrite;
};

//Handler for the constant e. e is converted to math.e

class eulerHandler : public MatchFinder::MatchCallback {
public:
   eulerHandler(Rewriter &Rewrite) : Rewrite(Rewrite)  {}

virtual void run(const MatchFinder::MatchResult &Results) {
    const clang::DeclRefExpr* eulerfinder =
Results.Nodes.getNodeAs<clang::DeclRefExpr>("EULER");
    Rewrite.ReplaceText(eulerfinder->getBeginLoc(), "math.e");
  }

private:
  Rewriter &Rewrite;
};

// Implementation of the ASTConsumer interface for reading an AST produced
// by the Clang parser. It registers a couple of matchers and runs them on
// the AST.
class MyASTConsumer : public ASTConsumer {
public:
  MyASTConsumer(Rewriter &R) : HandlerForIf(R), HandlerForFor(R),
HandlerForpinMode(R), HandlerForLoopExpr(R), HandlerForDelay(R),
HandlerForSetup(R), HandlerForCompoundStmt(R),
  HandlerForPower(R), HandlerForSqrt(R), HandlerForSin(R), HandlerForCos(R),
HandlerForTan(R), HandlerForDelayMicroseconds(R), HandlerForMillis(R),
HandlerForMicros(R), HandlerForPulseIn(R),
  HandlerForPinModePin(R), HandlerForINPUT(R), HandlerForOUTPUT(R),
HandlerForINPUTPULLUP(R), HandlerForIsAlpha(R),HandlerForIsAlphaVar(R),
HandlerForIsAlphaNumeric(R),
  HandlerForIsAlphaNumericVar(R), HandlerForIsAscii(R),
HandlerForIsAsciiVar(R), HandlerForIsDigit(R), HandlerForIsDigitVar(R),
HandlerForIsLowerCase(R), HandlerForIsLowerCaseVar(R),
   HandlerForIsPunct(R), HandlerForIsPunctVar(R), HandlerForIsSpace(R),
HandlerForIsSpaceVar(R), HandlerForIsUpperCase(R),
HandlerForIsUpperCaseVar(R), HandlerForIsWhitespace(R),
HandlerForIsWhitespaceVar(R),
   HandlerForAnalogRead(R), HandlerForAnalogWrite(R),
HandlerForDigitalRead(R), HandlerForDigitalWrite(R), HandlerForPi(R),
HandlerForEuler(R){
    // Add a simple matcher for finding 'if' statements.
    Matcher.addMatcher(ifStmt().bind("ifStmt"), &HandlerForIf);

    // Add a complex matcher for finding 'for' loops with an initializer set
    // to 0, < comparison in the codition and an increment. For example:
    //
    //  for (int i = 0; i < N; ++i). Just to test it out. Will change for to
for in range
    Matcher.addMatcher(
        forStmt(hasLoopInit(declStmt(hasSingleDecl(
                    varDecl(hasInitializer(integerLiteral(equals(0))))
                        .bind("initVarName")))),
                hasIncrement(unaryOperator(
                    hasOperatorName("++"),
                    hasUnaryOperand(declRefExpr(to(

varDecl(hasType(isInteger())).bind("incVarName")))))),
                hasCondition(binaryOperator(
                    hasOperatorName("<"),
                    hasLHS(ignoringParenImpCasts(declRefExpr(to(

varDecl(hasType(isInteger())).bind("condVarName"))))),
                    hasRHS(expr(hasType(isInteger()))))))
            .bind("forLoop"),
        &HandlerForFor);

//Add A matcher for PinMode

Matcher.addMatcher(
       callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("pinMode")))).bind("pinMode"),
&HandlerForpinMode);

//Add A matcher for void_loop function of Arduino

Matcher.addMatcher(
  functionDecl(isExpansionInMainFile(), hasName("loop"),
parameterCountIs(0)).bind("loopexpr"), &HandlerForLoopExpr);

 //Add A matcher for delay() function
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("delay")))).bind("delay"), &HandlerForDelay);

 //Add A matcher to delete Void Setup()
 Matcher.addMatcher(
   functionDecl(isExpansionInMainFile(), hasName("setup")).bind("setupfunc"),
&HandlerForSetup);

//Add A matcher to remove { } braces
Matcher.addMatcher(
  stmt(isExpansionInMainFile(), compoundStmt()).bind("compoundstmt"),
&HandlerForCompoundStmt);

//Add a matcher to convert power to math.pow
Matcher.addMatcher(
  stmt(isExpansionInMainFile(),
has(declRefExpr(throughUsingDecl(hasName("pow"))))).bind("pow"),
&HandlerForPower);

//Add a matcher to convert sqrt to math.sqrt
Matcher.addMatcher(
  stmt(isExpansionInMainFile(),
has(declRefExpr(throughUsingDecl(hasName("sqrt"))))).bind("sqrt"),
&HandlerForSqrt);

//Add a matcher to convert sin to math.sin
Matcher.addMatcher(
  stmt(isExpansionInMainFile(),
has(declRefExpr(throughUsingDecl(hasName("sin"))))).bind("sin"),
&HandlerForSin);

//Add a matcher to convert cos to math.cos
Matcher.addMatcher(
  stmt(isExpansionInMainFile(),
has(declRefExpr(throughUsingDecl(hasName("cos"))))).bind("cos"),
&HandlerForCos);

//Add a matcher to convert tan to math.tan
Matcher.addMatcher(
  stmt(isExpansionInMainFile(),
has(declRefExpr(throughUsingDecl(hasName("tan"))))).bind("tan"),
&HandlerForTan);

//Add a matcher to convert delayMicroseconds() to its Micropython equivalent.
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("delayMicroseconds")))).bind("delayMicroseconds")
, &HandlerForDelayMicroseconds);

//Add a matcher to convert millis() to its Micropython equivalent.
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("millis")))).bind("millis"), &HandlerForMillis);

//Add a matcher to convert micros() to its Micropython equivalent.
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("micros")))).bind("micros"), &HandlerForMicros);

  //Add a matcher to convert micros() to its Micropython equivalent.
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("pulseIn")))).bind("pulseIn"),
&HandlerForPulseIn);

  //Add a matcher to convert pin numbers to pin numbers with prefix 'p'
inside Pin.Mode.
Matcher.addMatcher(
  stmt(isExpansionInMainFile(),
hasAncestor(callExpr(callee(functionDecl(hasName("pinMode"))))),
has(integerLiteral())).bind("pinModePin"), &HandlerForPinModePin);

    //Add a matcher to convert INPUT to IN. pinmode uses Pin.Mode(PIN.IN)
Matcher.addMatcher(
  stmt(isExpansionInMainFile(),
declRefExpr(to(varDecl(hasName("INPUT"))))).bind("INPUT"), &HandlerForINPUT);

//Add a matcher to convert Output to OUT. pinmode uses Pin.Mode(PIN.OUT)
Matcher.addMatcher(
  stmt(isExpansionInMainFile(),
declRefExpr(to(varDecl(hasName("OUTPUT"))))).bind("OUTPUT"),
&HandlerForOUTPUT);

//Add a matcher to convert INPUT_PULLUP to PULLUP pinmode uses
Pin.Mode(PIN.PULL_UP)
Matcher.addMatcher(
  stmt(isExpansionInMainFile(),
declRefExpr(to(varDecl(hasName("INPUT_PULLUP"))))).bind("INPUT_PULLUP"),
&HandlerForINPUTPULLUP);

//Add a matcher to convert isAlpha to ure.match()
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("isAlpha")))).bind("isAlpha"),
&HandlerForIsAlpha);

//Add a matcher to add the regex string inside the isAlpha()
Matcher.addMatcher(
  declRefExpr(isExpansionInMainFile(), to(varDecl()),
hasAncestor(callExpr(callee(functionDecl(hasName("isAlpha")))))).bind("isAlph
aVar"), &HandlerForIsAlphaVar);

//Add a matcher to convert isAlphaNumeric to ure.match()
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("isAlphaNumeric")))).bind("isAlphaNumeric"),
&HandlerForIsAlphaNumeric);

//Add a matcher to add the regex string inside the isAlphaNumeric()
Matcher.addMatcher(
  declRefExpr(isExpansionInMainFile(), to(varDecl()),
hasAncestor(callExpr(callee(functionDecl(hasName("isAlphaNumeric")))))).bind(
"isAlphaNumericVar"), &HandlerForIsAlphaNumericVar);

//Add a matcher to convert isAscii to ure.match()
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("isAscii")))).bind("isAscii"),
&HandlerForIsAscii);

//Add a matcher to add the regex string inside the isAscii()
Matcher.addMatcher(
  declRefExpr(isExpansionInMainFile(), to(varDecl()),
hasAncestor(callExpr(callee(functionDecl(hasName("isAscii"))))).bind("isAsci
iVar"), &HandlerForIsAsciiVar);

//Add a matcher to convert isDigit to ure.match()
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("isDigit")))).bind("isDigit"),
&HandlerForIsDigit);

//Add a matcher to add the regex string inside the isDigit()
Matcher.addMatcher(
  declRefExpr(isExpansionInMainFile(), to(varDecl()),
hasAncestor(callExpr(callee(functionDecl(hasName("isDigit"))))).bind("isDigi
tVar"), &HandlerForIsDigitVar);

//Add a matcher to convert isLowerCase to ure.match()
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("isLowerCase")))).bind("isLowerCase"),
&HandlerForIsLowerCase);

//Add a matcher to add the regex string inside the isLowerCase()
Matcher.addMatcher(
  declRefExpr(isExpansionInMainFile(), to(varDecl()),
hasAncestor(callExpr(callee(functionDecl(hasName("isLowerCase"))))).bind("is
LowerCaseVar"), &HandlerForIsLowerCaseVar);

//Add a matcher to convert isPunct to ure.match()
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("isPunct")))).bind("isPunct"),
&HandlerForIsPunct);

//Add a matcher to add the regex string inside the isPunct()
Matcher.addMatcher(
  declRefExpr(isExpansionInMainFile(), to(varDecl()),
hasAncestor(callExpr(callee(functionDecl(hasName("isPunct"))))).bind("isPunc
tVar"), &HandlerForIsPunctVar);

//Add a matcher to convert isSpace to ure.match()
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("isSpace")))).bind("isSpace"),
&HandlerForIsSpace);

//Add a matcher to add the regex string inside the isSpace()
Matcher.addMatcher(
  declRefExpr(isExpansionInMainFile(), to(varDecl()),
hasAncestor(callExpr(callee(functionDecl(hasName("isSpace"))))).bind("isSpac
eVar"), &HandlerForIsSpaceVar);
```

```
//Add a matcher to convert isUpperCase to ure.match()
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("isUpperCase")))).bind("isUpperCase"),
&HandlerForIsUpperCase);

//Add a matcher to add the regex string inside the isUpperCase()
Matcher.addMatcher(
  declRefExpr(isExpansionInMainFile(), to(varDecl()),
hasAncestor(callExpr(callee(functionDecl(hasName("isUpperCase")))))).bind("is
UpperCaseVar"), &HandlerForIsUpperCaseVar);

//Add a matcher to convert isWhitespace to ure.match()
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("isWhitespace")))).bind("isWhitespace"),
&HandlerForIsWhitespace);

//Add a matcher to add the regex string inside the isWhitespace()
Matcher.addMatcher(
  declRefExpr(isExpansionInMainFile(), to(varDecl()),
hasAncestor(callExpr(callee(functionDecl(hasName("isWhitespace")))))).bind("i
sWhitespaceVar"), &HandlerForIsWhitespaceVar);
//Add a matcher to convert analogRead to its micropython equivalent.
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("analogRead")))).bind("analogRead"),
&HandlerForAnalogRead);

//Add a matcher to convert analogWrite to its micropython equivalent.
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("analogWrite")))).bind("analogWrite"),
&HandlerForAnalogWrite);

//Add a matcher to convert digitalRead to its micropython equivalent.
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("digitalRead")))).bind("digitalRead"),
&HandlerForDigitalRead);

//Add a matcher to convert digitalWrite to its micropython equivalent.
Matcher.addMatcher(
  callExpr(isExpansionInMainFile(),
callee(functionDecl(hasName("digitalWrite")))).bind("digitalWrite"),
&HandlerForDigitalWrite);

//Add a matcher to convert the constant Pi to its micropython equivalent.
Matcher.addMatcher(
  declRefExpr(isExpansionInMainFile(),
to(varDecl(hasName("PI")))).bind("PI"), &HandlerForPi);

//Add a matcher to convert the constant e to its micropython equivalent.
Matcher.addMatcher(
  declRefExpr(isExpansionInMainFile(),
to(varDecl(hasName("EULER")))).bind("EULER"), &HandlerForEuler);
```

```
  }

  void HandleTranslationUnit(ASTContext &Context) override {
    // Run the matchers when we have the whole TU parsed.
    Matcher.matchAST(Context);

  }

private:
  IfStmtHandler HandlerForIf;
  IncrementForLoopHandler HandlerForFor;
  pinModeVariableHandler HandlerForpinMode;
  loopExprHandler HandlerForLoopExpr;
  delayHandler HandlerForDelay;
  setupHandler HandlerForSetup;
  compoundStmtHandler HandlerForCompoundStmt;
  powerHandler HandlerForPower;
  sqrtHandler HandlerForSqrt;
  sinHandler HandlerForSin;
  cosHandler HandlerForCos;
  tanHandler HandlerForTan;
  delayMicrosecondsHandler HandlerForDelayMicroseconds;
  millisHandler HandlerForMillis;
  microsHandler HandlerForMicros;
  pulseInHandler HandlerForPulseIn;
  pinModePinHandler HandlerForPinModePin;
  inputHandler HandlerForINPUT;
  outputHandler HandlerForOUTPUT;
  inputpullupHandler HandlerForINPUTPULLUP;
  isAlphaHandler HandlerForIsAlpha;
  isAlphaVarHandler HandlerForIsAlphaVar;
  isAlphaNumericHandler HandlerForIsAlphaNumeric;
  isAlphaNumericVarHandler HandlerForIsAlphaNumericVar;
  isAsciiHandler HandlerForIsAscii;
  isAsciiVarHandler HandlerForIsAsciiVar;
  isDigitHandler HandlerForIsDigit;
  isDigitVarHandler HandlerForIsDigitVar;
  isLowerCaseHandler HandlerForIsLowerCase;
  isLowerCaseVarHandler HandlerForIsLowerCaseVar;
  isPunctHandler HandlerForIsPunct;
  isPunctVarHandler HandlerForIsPunctVar;
  isSpaceHandler HandlerForIsSpace;
  isSpaceVarHandler HandlerForIsSpaceVar;
  isUpperCaseHandler HandlerForIsUpperCase;
  isUpperCaseVarHandler HandlerForIsUpperCaseVar;
  isWhitespaceHandler HandlerForIsWhitespace;
  isWhitespaceVarHandler HandlerForIsWhitespaceVar;
  analogReadHandler HandlerForAnalogRead;
  analogWriteHandler HandlerForAnalogWrite;
  digitalReadHandler HandlerForDigitalRead;
  digitalWriteHandler HandlerForDigitalWrite;
  piHandler HandlerForPi;
  eulerHandler HandlerForEuler;


  MatchFinder Matcher;
```

```
};

// For each source file provided to the tool, a new FrontendAction is
created.
class MyFrontendAction : public ASTFrontendAction {
public:
  MyFrontendAction() {}
  void EndSourceFileAction() override {
   SourceManager &SM = TheRewriter.getSourceMgr();
   llvm::errs() << "** EndSourceFileAction for: "
                 << SM.getFileEntryForID(SM.getMainFileID())->getName() <<
"\n";
//Now emit the Rewritten Buffer
    TheRewriter.getEditBuffer(TheRewriter.getSourceMgr().getMainFileID())
        .write(llvm::outs());

std::error_code error_code;
        llvm::raw_fd_ostream outFile("output.txt", error_code,
llvm::sys::fs::F_None);
     TheRewriter.getEditBuffer(SM.getMainFileID()).write(outFile); // -->
this will write the result>
    outFile.close();

  }

  std::unique_ptr<ASTConsumer> CreateASTConsumer(CompilerInstance &CI,
                                                 StringRef file) override {
    TheRewriter.setSourceMgr(CI.getSourceManager(), CI.getLangOpts());
    return std::make_unique<MyASTConsumer>(TheRewriter);
  }

private:
  Rewriter TheRewriter;
};

int main(int argc, const char **argv) {
  CommonOptionsParser op(argc, argv, MatcherSampleCategory);
  ClangTool Tool(op.getCompilations(), op.getSourcePathList());

  return Tool.run(newFrontendActionFactory<MyFrontendAction>().get());
}
```

Installation Instructions

To run the tool you require a full install of both Clang/LLVM and Libtooling along with the ninja build system. Any other linker will not work, and ninja is more performant. You can use the instructions on this page. The download+build process may take anywhere from 1.5 hours to 5+ hours depending on your computer, and may consume upto 16 GB of RAM.

Using the -DCMAKE_BUILD_TYPE=Release flag in the CMAKE step is helpful in speeding up the process.

1.) Clone this repository.

2.) Place the micropy-convert folder inside the clang-tools-extra directory:

```
$ cd clang-llvm/llvm-project/clang-tools-extra
```

3.) Place the Arduino-headerfiles folder inside the clang directory:

```
$ cd clang-llvm/llvm-project
```

4.) Build the tool by running ninja from inside the build directory:

```
$ cd clang-llvm/llvm-project/build
$ ninja
```

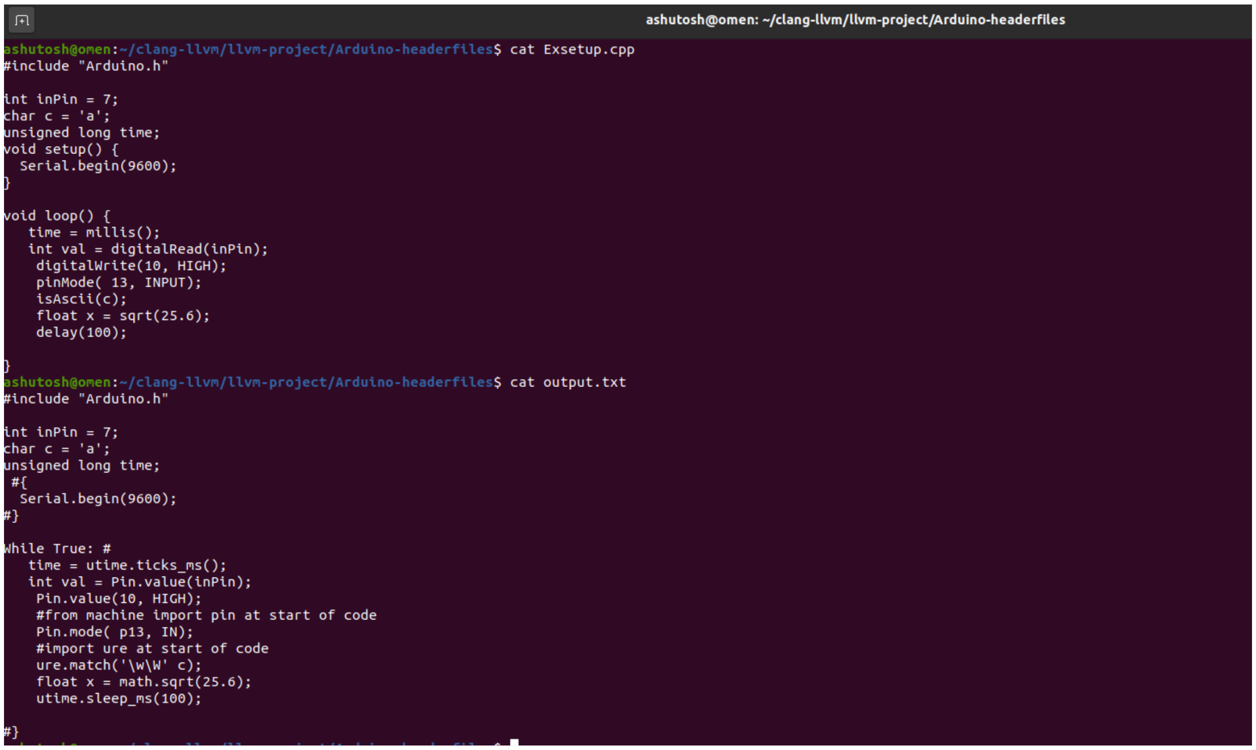
Using the tool

Place the file which has to be translated into the Arduino-headerfiles folder. Make sure it has a .cpp extension and has #include "Arduino.h" header. This forces the tool to use our modified header files. Then type:

```
$ ~/clang-llvm/llvm-project/build/bin/micropy-convert FILENAME.cpp --
```

The converted output will be visible on the terminal, as well as an output.txt file located within the same folder.

For more information on how to modify and build the tool with more nodes, read Report.md



```
ashutosh@omen: ~/clang-llvm/llvm-project/Arduino-headerfiles
ashutosh@omen:~/clang-llvm/llvm-project/Arduino-headerfiles$ cat Exsetup.cpp
#include "Arduino.h"

int inPin = 7;
char c = 'a';
unsigned long time;
void setup() {
  Serial.begin(9600);
}

void loop() {
   time = millis();
   int val = digitalRead(inPin);
    digitalWrite(10, HIGH);
    pinMode( 13, INPUT);
    isAscii(c);
    float x = sqrt(25.6);
    delay(100);

}
ashutosh@omen:~/clang-llvm/llvm-project/Arduino-headerfiles$ cat output.txt
#include "Arduino.h"

int inPin = 7;
char c = 'a';
unsigned long time;
 #{
  Serial.begin(9600);
#}

While True: #
   time = utime.ticks_ms();
   int val = Pin.value(inPin);
    Pin.value(10, HIGH);
    #from machine import pin at start of code
    Pin.mode( p13, IN);
    #import ure at start of code
    ure.match('\w\W' c);
    float x = math.sqrt(25.6);
    utime.sleep_ms(100);

#}
```